

ARI Contractor Report 2005-11

Adaptive Instructional Systems

Paul Cobb

Cybernet Systems Corporation

Best Available Copy

This report is published to meet legal and contractual requirements and may not meet ARI's scientific or professional standards for publication.

September 2005

**United States Army Research Institute
for the Behavior and Social Sciences**

20050928 000

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1. REPORT DATE (dd-mm-yy) September 2005		2. REPORT TYPE Final		3. DATES COVERED (from... to) January 2000 – July 2000	
4. TITLE AND SUBTITLE Adaptive Instructional Systems				5a. CONTRACT OR GRANT NUMBER DASW01-00-M-4088	
				5b. PROGRAM ELEMENT NUMBER 20262785	
6. AUTHOR(S) Paul Cobb (Cybernet Systems Corporation)				5c. PROJECT NUMBER A790	
				5d. TASK NUMBER 206	
				5e. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Cybernet Systems Corporation 727 Airport Boulevard Ann Arbor, MI 48108-1639				8. PERFORMING ORGANIZATION REPORT NUMBER Final Report 331	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U. S. Army Research Institute for the Behavioral & Social Sciences ATTN Rotary Wing Aviation Research Unit 2511 Jefferson Davis Highway Arlington, VA 22202				10. MONITOR ACRONYM ARI	
				11. MONITOR REPORT NUMBER Contractor Report 2005-11	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES Contracting Officer's Representative and Subject Matter POC: William R. Howse. This report is published to meet legal and contractual requirements and may not meet ARI's scientific and/or professional standards for publication.					
14. ABSTRACT (Maximum 200 words): This report was developed under SBIR contract for Topic OSD99-004. This report describes the Phase I activities conducted for the Army Research Institute (ARI) at Cybernet Systems Corporation during the period of January 24 th , 2000 to July 24 th , 2000, under the "Adaptive Instructional Systems" contract DASW01-00-M-4088. These activities focused on four major areas: <ol style="list-style-type: none"> 1. Develop a Helicopter Flight Model for the Model-Based Reasoning Diagnostic Engine (MBRDE), 2. Integrate the Model-Based Reasoning Diagnostic Engine into the OpenSkies Virtual Environment Training System, 3. Enhance the OpenSkies Virtual Environment Training System to Focus the Student's Effort in Deficient Areas, and 4. Demonstrate Adaptive Training by Creating a Scenario for Hovering a Helicopter in OpenSkies, <p>This research demonstrated that an Army adaptive instructional system can be effectively developed and implemented.</p>					
15. SUBJECT TERMS Event-Based Training, Adaptive Training, Artificial Intelligence, Virtual Reality, Intelligent Tutoring, Training Technology					
SECURITY CLASSIFICATION OF			19. LIMITATION OF ABSTRACT	20. NUMBER OF PAGES	21. RESPONSIBLE PERSON
16. REPORT Unclassified	17. ABSTRACT Unclassified	18. THIS PAGE Unclassified	Unlimited		Ellen Kinzer Technical Publication Specialist 703-602-8047

Standard Form 298

Table of Contents

1. Introduction.....	1
1.1 Motivation.....	1
1.2 Project Goals.....	1
1.3 Project Methodology.....	2
1.4 Project Outcomes.....	2
1.5 Developed System.....	4
1.6 Commercialization.....	4
2. Phase I Work.....	5
2.1 Intelligent Systems.....	5
2.2 Model-Based Systems Overview.....	7
2.3 Diagnostic Model-Based Systems.....	7
2.4 Knowledge Used for Model-Based Diagnosis.....	9
2.5 Model Accuracy.....	9
2.6 Model-Based System Design.....	10
2.7 Advantages of Model-Based Diagnostic Systems.....	11
2.8 Disadvantages of Model-Based Diagnostic Systems.....	11
2.9 Examples of Model-Based Diagnosis.....	11
2.10 Cybernet's Approach to a Model-Based System.....	13
3. Model-Based Reasoning Diagnostic Engine.....	15
3.1 Anytime Algorithms.....	15
3.2 Anytime Diagnosis.....	16
3.3 Hierarchical Analysis.....	21
3.4 Internal Model Construction.....	22
3.5 The Input.....	22
3.6 The Tree.....	22
3.7 The Set of Graphs.....	23
3.8 The Expansion of the Input to Create the Model.....	24
3.9 Graph Construction Algorithm.....	25
3.10 Probability Percolation Algorithm.....	25

3.11 Drawing Conclusions In Real Time.....	26
3.12 Possible Cause Lists.....	26
3.13 Possible Effect Lists.....	27
3.14 Minimal Covers	27
3.15 Overview of Algorithm to Find Minimal Covers	28
4. Learning Styles.....	31
4.1 Diagnosing a Student's Learning Style.....	32
4.2 Physical Stimuli	32
4.3 Perceptual.....	32
4.4 Environmental Stimuli.....	33
5. OpenSkies Virtual Environment Training System.....	34
5.1 Why is it needed?.....	34
5.2 How will this system improve the student's performance?	34
5.3 What are the advantages of this system?	35
5.4 What is unique about this system?.....	35
5.5 The OpenSkies Scenario Development and Performance Measurement.....	35
5.6 Scenario Development.....	36
5.7 Performance Analysis	37
5.8 Example Scenario Development.....	38
5.9 Running the Example Scenario.....	40
5.10 Addressing Student Errors	43
5.11 Example of Branching using Activation and Deactivation.....	47
6. Future Directions	49
7. Marketing Plan.....	50
7.1 Target Market.....	51
7.2 Product	51
7.3 Sales	51
7.4 Budget	51
8. Conclusions.....	51

Table of Figures

Figure 1 Simple Model for Helicopter Flight	6
Figure 2: Observations and Predictions in Model-Based Reasoning.....	8
Figure 3: An Adaptive Feedback Collective Component	14
Figure 4: Initial Representation.....	17
Figure 5: Data Origins.....	17
Figure 6: Initial Data Association Lists	17
Figure 7: Initial Data-Component Associations.....	17
Figure 8: Updated Data Association Lists	18
Figure 9: Second Data-Component Associations	18
Figure 10: First Component Grouping.....	18
Figure 11: Initial Component Hierarchy	18
Figure 12: Final Data Association Lists.....	18
Figure 13: Final Representation.....	18
Figure 14: Final Component Hierarchy	19
Figure 15: Example Graph (for Illustrating the Minimal Cover Computational Algorithm)	29
Figure 16: Scenario Edit Window.....	38
Figure 17: Instruction for Student by Time.....	40
Figure 18: Control Panel.....	41
Figure 19: Evaluation Results.....	41
Figure 20: Mission Evaluation.....	42
Figure 21: Event Activation Window	45
Figure 22: Event Deactivation Window.....	46
Figure 23: Key Input Window	48
Figure 24: Summary of Sales for Leading Flight Simulators and On-line Games	50

1. Introduction

This report describes the Phase I activities conducted for the Army Research Institute (ARI) at Cybernet Systems Corporation during the period of January 24th, 2000 to July 24th, 2000, under the "Adaptive Instructional System," contract DASW01-00-M-4088. These activities focused on four major areas:

- 1) Develop a Helicopter Flight Model for the Model-Based Reasoning Diagnostic Engine (MBRDE),
- 2) Integrate the Model-Based Reasoning Diagnostic Engine into the OpenSkies Virtual Environment Training System,
- 3) Enhance the OpenSkies Virtual Environment Training System to Focus the Student's Effort in Deficient Areas,
- 4) Demonstrate Adaptive Training by Creating a Scenario for Hovering a Helicopter in OpenSkies,

This research demonstrated that an Army adaptive instructional system can be effectively developed and implemented.

1.1 Motivation

This Phase I research was important because it facilitated the development of a model-based adaptive instruction system for teaching helicopter pilots. An adaptive training system that focuses on the deficiencies of the student can both increase the number of students meeting the minimum proficiency levels and save training time. Cybernet plans to continue our development of this adaptive training methodology based on expert system and model-based techniques. This will allow us to determine where a student's deficiencies lie and adapt the training facility to focus on those problem areas. Unlike other simulators that are currently available, we will be incorporating each student's learning style preference into the simulation instruction. The proposed Phase II effort is focused on continued development and enhancement of this system. Our Phase I work demonstrated the feasibility of accomplishing this through the integration of our Model-Based Reasoning Diagnostic Engine and our OpenSkies Virtual Environment to produce a complete Adaptive Training System.

1.2 Project Goals

The intent of this effort was to develop a technology demonstration for an adaptive training system, focused on diagnosis of student behavior. In the Phase I effort Cybernet proposed building a diagnostic engine for student fault detection and remediation. In this phase of the effort three distinct components were defined. The first was an adaptation of Cybernet's *Data Collection and Analysis Environment (DCAE)*, which was used early in the project to provide the data framework for the limit checking and expert systems architectures. Expert systems require data stored in a "blackboard" system. The DCAE

provides this capability over a distributed network of data collection, storage, and routing processes.

The second architecture was for data pre-processing. The baseline approach was to browse data in the "blackboard" for limit ranges. For each datum, a low and high range was checked. The Boolean effect for each datum was "Out-of-Range-Low", "Out-of-Range-High", or "In-range". Either "Out-of-Range-Low" or "Out-of-Range-High" represented a fault cue.

1.3 Project Methodology

The entire Phase I project consisted of four phases:

- 1) The project startup phase, which provided background information on adaptive instructional systems, learning styles, and model-based reasoning systems
- 2) The design stage, which concentrated on creating model representations, a learning procedure, and a system architecture.
- 3) The development stage, which concentrated on the compilation of a proof-of-concept adaptive instructional system.
- 4) The planning stage, which used the information obtained from the Phase I research to develop a plan for utilizing the technology in the proposed system.

1.4 Project Outcomes

The Phase I project has successfully completed the project goals established above. We have achieved these goals by concentrating on the incremental Phase I objectives. The Phase I objectives and how they have been accomplished are outlined below:

- 1) **Develop Helicopter Flight Model for Model-Based Reasoning Diagnostic Engine (MBRDE).** We have researched and developed a model describing the flight model for hovering a helicopter. This model accesses the helicopter operating parameters as an input to the diagnostic engine. Since the OpenSkies System already contains a well-developed model for the TH-57 helicopter, we used this flight model for this development effort. This step also defines the possible errors encountered in hovering the helicopter.

The helicopter model in OpenSkies consists of three high-level systems: the landing gear, the engine, and the rotors. The landing gear simply represents the helicopter's interaction with the ground, and is modeled using a basic spring/damper system. The engine system is also straightforward, and represents the Bell JetRanger's three throttle settings (off, idle, and open), which in turn controls the rotor rotational velocity. Fuel (and its weight) also is a part of the engine system, as well as accompanying fuel use that depends on the engine RPM and environmental factors. The most robust system is that which models the rotors. Both the main and tail rotors are modeled complexly to allow for factors such as wind speed, ground effect, rotor torque, available engine power, and helicopter velocity.

The controls of the helicopter work as a real helicopter: the rudder affects the tail rudder, the cyclic differentially pitches the blades of the main rotor, and the collective modifies the pitch of all the blades of the main rotor. The end effect is that the model for a helicopter in OpenSkies is complex enough to represent piloting requirements such as in-ground-effect to out-of-ground-effect transitions, changing rudder requirements depending on collective, and forward blade stalls. However, the model also has the ability to have artificial controls applied to it to simplify operation. For instance, having the model automatically adjust the rudder, maintain a hover, or increase collective with changing cyclic. This creates a helicopter model that runs from completely autopilot controlled to completely pilot controlled.

- 2) **Integrate Model-Based Reasoning Diagnostic Engine into the OpenSkies Virtual Environment Training System.** This task entailed converting the MBRDE into a dynamically linked library for inclusion into the OpenSkies system. This task has been completed. We have already adapted the OpenSkies system in order to parse the data as input to the Model-Based Reasoning Diagnostic Engine. This system is based on of OpenSkies High Level Architecture (HLA) Interface.
- 3) **Enhance OpenSkies Virtual Environment Training System to Focus the Student's Effort in Deficient Areas.** This task defines some of the adaptive capabilities of the OpenSkies system. We have modified the scenario scripting capability to handle branching to other parts of the scenario as well as adding capability for answering questions and providing tutorials. Further, we have designed new capabilities for more interactive capabilities in the simulation. This will allow us to quickly redefine new input data from outside sources for creating new types of scenarios.
- 4) **Demonstrate Adaptive Training by Creating a Scenario for Hovering a Helicopter in OpenSkies.** We have developed a specific scenario for hovering a helicopter that includes branches to earlier parts of the scenario, question and answers and a tutorial for hovering the helicopter.
- 5) **Integrate System into OpenSkies Simulator.** For this task, we integrated the adaptive instructional system into the OpenSkies simulator. Demonstrating such integration illustrated how the adaptive instruction system can be used as a human computer interface, and helps us design the Phase II system. We specifically developed a basic system for training on a virtual helicopter.
- 6) **Produce a Final Technical Report.** This report completes this task. The goal of this task was to fully document the results of the project. We will use this information to arrive at a complete design and methodology for the Phase II system. It will include recommendations for hardware components, a mapping of appropriate algorithms to this hardware, and an analysis of the proposed system's capabilities. It will also contain a concept of operation, course level parts descriptions, and estimated level of software/firmware development, integration, and maintenance.

The following section provides details of specific aspects of the automatic learning system. Meeting all of these objectives has resulted in a fundamental understanding of the issues and solutions in performing recognition of dynamic and static gestures on inexpensive personal computer platforms. We will use our understanding acquired during this Phase I to address the integration of the adaptive instruction system into the military's system.

1.5 Developed System

As detailed in Section 2, our system takes a teaching model created by the instructor and a model designed around the student to create a final instructional model. The system is used as follows:

1. The instructor determines what aspects of flight simulation he would like to teach and develops an instructional model for the given scenario.
2. The student begins to fly the scenario while the simulator begins to create a model about him and his actions.
3. The simulation generates remedial feedback for the student in order to improve his performance.
4. The simulator updates the student model as the student trains.
5. Steps 3 & 4 repeat until training is finished.

1.6 Commercialization

The proposed technology will be leveraged into Cybernet's OpenSkies Massive Multiplayer training and gaming simulation business. Cybernet has developed a massively multi-player simulation technology for air, sea and land game and simulation play¹. While this technology was originally developed for low cost government simulation for training, the Company plans to adapt the technology to revolutionize the consumer network gaming and flight simulator industry. The Company plans to distribute its OpenSkies simulation products at retail into the market, which is currently defined by Microsoft Flight Simulator, ProPilot, and Flight Unlimited. To make a significant inroad to this market, Cybernet plans to sell the product not as an end to itself, but as the entry point to a new game playing experience.

Cybernet plans to revolutionize the gaming industry by coupling the experience of leading military commanders with the on-line flight gaming experience. The military commander will structure a training process which 'recruits' players, gives them flight training modeled after current military doctrine, and then leads graduates in a multi-player interactive war game. Because the technology is compatible with Government training

¹ The complete business plan is available upon request.

needs Cybernet plans to sell this product into the Government space for integration and training purposes as well.

2. Phase I Work

In addition to the original Phase I Work Plan items, we have extended our work to gain a better understanding of student learning styles. This work has allowed us to develop a working Phase II concept that will be adaptive to many different types of students.

Learning styles are defined as the composite of characteristic cognitive, affective, and physiological factors that serve as relatively stable indicators of how a learner perceives, interacts with, and responds to the learning environment (Keefe, 1979). Included in this definition are "cognitive styles," which are intrinsic information-processing patterns that represent a person's typical mode of perceiving, thinking, remembering and problem solving (Messick, 1969). While there are over 250 conceptually distinct approaches to instruction (Parloff, 1980) all vying for distinction as the most effective, we will analyze the foremost models, and incorporate them into a model that addresses a military audience.

We have also extended our research on adaptive model-based reasoning. This research will allow us to enhance our current Model-Based Reasoning Diagnostic Engine to create a more adaptive system.

The MBRDE framework provides a generic approach to diagnosis of learner behavior, a task that is often considered to be too complex to be cost-effective. The techniques developed facilitate model-based reasoning about the learner's problem solving behavior on the sole basis of a qualitative simulation model. Hence, by only specifying the input for the qualitative simulator, a hierarchical set of domain models can be automatically generated.

The diagnostic process traces errors made in individual reasoning steps. This focus on errors in the learner's problem solving behavior strongly influences the educational approach of the system. People learn from their errors. This view on education is different from the one underlying most diagnostic approaches. Instead of focusing on tracing misconceptions in the learner's knowledge that can be remediated, we support the learner's capability of self-repair.

2.1 Intelligent Systems

From their inception, intelligent systems have been applied to the task of automated diagnosis. Diagnostic expert systems were among the first successes of artificial intelligence, and played a key role in the development of the field.

As the devices under diagnosis grew more complex, diagnostic system programmers could no longer be assured their *a priori* knowledge of how the devices might fail would be complete. It became apparent that expert systems' brittleness -- in the case of diagnosis, their inability to diagnose novel faults -- would limit their usefulness. In order

to diagnose these unpredicted (or "unknown") faults, a more generalized diagnostic approach was needed. This need led to the application of the *model-based reasoning* paradigm to the task of automated device diagnosis.

The complexity of the diagnostic task has increased in ways beyond the number of components and interconnections in devices under diagnosis. Intelligent systems are called upon to provide diagnoses in applications where time is an increasingly critical resource, ranging from nuclear plant control to student health and status monitoring. In such real-time applications where operators must be notified of detected anomalies and their probable cause(s) in a timely manner, model-based diagnostic systems are faced with a tradeoff between *diagnostic speed and detail*. Simply put, the more accurate and detailed the device model, the more time will be required to reach a diagnostic conclusion.

Noting the similarity between this diagnostic tradeoff and that with which real-time planning systems must contend, we have looked to *anytime algorithms* for inspiration. This led to the development of a model-based diagnostic framework that allows a diagnostic system to produce a diagnostic response at any time, with the response becoming increasingly accurate as more processing time is allocated to the diagnostic task.

Cybernet's Model-Based Diagnostic Engine was originally developed under an Air Force contract to detect and diagnose student behavioral anomalies. We intend to use this model-based system to define a training model for helicopter flight that will allow us to determine in real-time where the student is making errors and adapt the training system to focus on these errors. A simple model of the helicopter flight can be defined as follows.

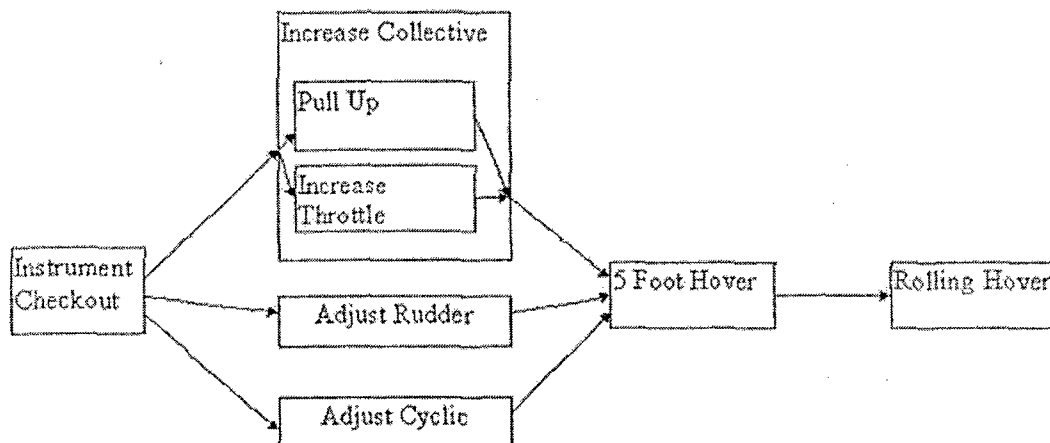


Figure 1: Simple Model for Helicopter Flight

If an instructor were required to develop an expert system that checks for each individual case, the training system could possibly take weeks or even months to develop. Instead we have the instructor develop a simple relational model of the flow of proper flight of the helicopter. The student follows the flow of model and at any time that the student has incorrectly operated the helicopter, the *diagnostic engine* will determine where the error was by tracing back through the model. The diagnostic engine is informed of an error in the flight by the OpenSkies engine that is watching for missed events in the scenario. The diagnostic engine then polls the simulator and traces through the model to determine where the student made an error. At this point, the training system will then be able to present the different options to the student based on the type of error.

For example, OpenSkies recognizes that the student did not make the Rolling Hover. The Model-Based Reasoning diagnostic Engine (MBRDE) then traces back through the system and determines that the 5 ft. Hover was most likely not properly executed. The system then looks for faults in the Increase Collective, Adjust Rudder and Adjust Cyclic tasks. Assuming no faults are found in Adjust Rudder and Adjust Cyclic and a fault was found in Increase Collective, the system assumes an error either in Pulling Up or Increasing the Throttle. The system checks for faults in each of these and finds that the pilot produced too much throttle.

At this point the MBRDE informs OpenSkies that the problem is in the Increase Throttle section of the Increase Collective and the system can then present the student with several options. The system could let the student ask a simple question, ask for a tutorial or point out to the student that the gauge is working improperly.

This research will allow Cybernet to develop this adaptive training system and incorporate it into our OpenSkies virtual reality environment for both demonstration of the technology and for eventual commercialization of the product.

2.2 Model-Based Systems Overview

For this proposal, we will talk about *component-oriented systems*. Such systems are concerned primarily with the structure of the artifact in question. An example of a component-oriented system is ENVISION [de Kleer & Brown, 1984]. ENVISION begins with a description of the physical structure of a device in terms of physically disjoint components and conduits connecting those components, a set of behavior rules for each component, and an input force applied to the device. The system then seeks to predict the behavior of an entire device as a sequence of possible future states along with complete causal analysis for the behavior.

2.3 Diagnostic Model-Based Systems

The basic paradigm of model-based reasoning for diagnosis can best be described as the interaction of observation and prediction [Davis and Hamacher, 1992]. Observations of the system being monitored indicate what the device is actually doing, and predictions based on an internal model of the system indicate what it is supposed to be doing. Model-based reasoning is based on knowledge of the structure and behavior of the device under observation. Model-based systems include a model of the device to be diagnosed,

which is then used to identify the cause(s) of the device's failure. The systems essentially perform internal simulations that generate predictions of how the device should behave. The systems also observe the device's actual behavior, and compare these observations with their own predictions. [Davis & Hamscher, 1992] depicts the interaction between prediction and observation that characterizes model-based diagnosis.

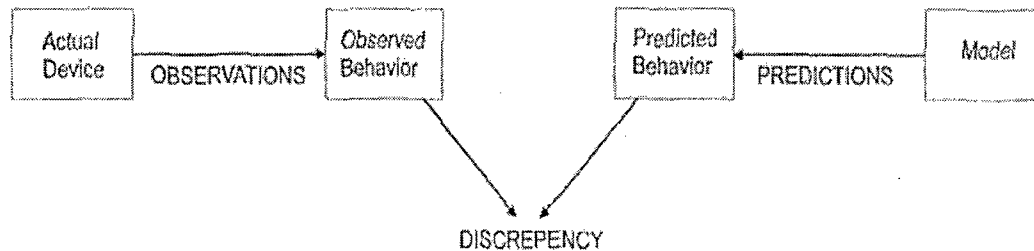


Figure 2: Observations and Predictions in Model-Based Reasoning

Any differences between prediction and observation indicate possible faults. As described in [Davis and Hamscher], the diagnosis task is one of determining which of the device's components could have failed in a way that accounts for all of the discrepancies observed. Because model-based systems draw their conclusions based on knowledge of a device's behavior and its structure, they are often said to reason from "first principles". This view of model-based reasoning is developed in [Reiter, 1987].

Three key points regarding model-based diagnostic systems are set forth in the introductory chapter of [Hamscher et al., 1992]:

1. *Knowledge about the internal structure and behavior of a designed artifact can be used to diagnose that artifact.*
2. *Model-based diagnosis programs generate, test, and discriminate diagnoses.*
3. *Modeling is the hard part of model-based diagnosis.*

The principles of model-based reasoning are well suited for performing device/system diagnoses. Model-based reasoning has become one of the most effective means of performing automated diagnosis on systems or artifacts. There are three key issues in the application of model-based approaches to the diagnosis of engineered artifacts:

- The knowledge used to drive the diagnosis;
- The diagnostic procedure;
- Modeling the real-world artifact(s).

2.4 Knowledge Used for Model-Based Diagnosis

Traditional diagnostic methods, ranging from fault trees to expert systems, have as their foundation an enumeration of possible faults, or, the way things might fail. Model-based diagnosis presents a significant departure from this line of thinking. The foundation of model-based diagnosis is the principle that knowledge about the internal structure and behavior of an artifact can be used to diagnose that system. This approach first appeared in the INTER [de Kleer, 1976] and SOPHIE [Brown et al., 1982] programs, which were both designed to perform troubleshooting of electronics systems.

Steps in Performing Model-Based Diagnosis

Model-based diagnosis can be broken down into three distinct steps, as described in [Davis and Hamscher, 1988]:

1. *Hypothesis generation.* Given a discrepancy between the way an artifact is operating and the way it ought to be behaving, the first task is to determine which components could have failed in such a way as to create the observed discrepancy. Among the techniques used to generate a list of candidate components are tracing through an internal representation of the system structure beginning from the location where the discrepancy is noticed and using information from multiple discrepancies to constrain the generation of the hypothesis list.
2. *Hypothesis testing.* Once the list of candidate hypotheses has been generated, each candidate component must be tested to see if can account for any or all of the observed discrepancies. Among the techniques used to perform hypothesis testing are fault-model simulations and constraint suspension.
3. *Hypothesis discrimination.* Finally, the diagnostic system must make some distinction between those hypotheses that survive the test stage. This process involves gathering additional information about the device, through such techniques as probing (making additional observations) or testing (changing inputs to the system and make observations in the new situation).

Diagnostic algorithms employed to perform the specific diagnostic reasoning in model-based diagnostic systems are based on standard AI techniques, and include:

- Theorem proving;
- Heuristic search;
- Qualitative simulation;
- Bayesian networks.

2.5 Model Accuracy

The most critical issue in model-based diagnosis is correctly modeling the artifact under diagnosis. Inherent in the use of a model for diagnostic reasoning is the implicit

assumption that the model is correct, and therefore that all discrepancies between prediction and observation can be traced back to faults in the device. As stated in [Davis and Hamscher, 1988], "The assumption that the model is correct is in fact *necessarily wrong in all cases*. It is wrong in ways that are sometimes quite obvious and sometimes quite subtle. Simply put, a model is a model precisely because it is not the device itself and hence must in many ways be only an approximation. There will always be things about the device that the model does not capture."

A significant challenge in defining artifact models is determining the correct level of abstraction at which to model the artifact. The selection of the models' 'level' involves a tradeoff between the accuracy of the model and its computational complexity. Simply put, the more detailed the models; the harder (and slower) they are to work with.

General-purpose models are constructed using standard AI technologies such as:

- Predicate logic;
- Frames;
- Constraints;
- Rules.

2.6 Model-Based System Design

Model-based systems typically use:

1. Observations of the device, typically observations at its inputs and outputs.
2. A description of the device's internal structure, typically a listing of its components and their interconnections.
3. A description of the behavior of each component.

These components then interact previously described to permit the system to detect discrepancies and to then generate, test, and discriminate its hypotheses as to what might be causing the discrepancies. There is no standard, or typical, configuration for model-based systems. A wide variety of AI techniques are used to create model-based diagnosis systems, including:

- Knowledge Representation:
 - predicate logic
 - production rules
 - slot-and-filler
 - frames
 - semantic nets
 - constraints
- Inference Engine:

- theorem proving
- heuristic search
- qualitative simulation
- Bayesian networks

Discussions of these AI techniques can be found in [Bundy, 1990], [Rich & Knight, 1991] and [Shapiro, 1987].

2.7 Advantages of Model-Based Diagnostic Systems

- Among the many advantages of the model-based approach to implementing diagnostic systems are the following:
- General reasoning scheme eliminates need for specific expert knowledge
- Device independent
- Works from an information source (the design) typically available when the device is first manufactured
- Can be less costly, since the model is often supplied by the description used to design and build the device
- Avoids the data acquisition bottleneck associated with expert knowledge capture and engineering
- Explanatory capability is inherent in the paradigm

2.8 Disadvantages of Model-Based Diagnostic Systems

Perhaps the greatest disadvantage of the model-based approach is summed up in the third main point taken from [Hamscher et al., 1992]: modeling is the hard part of model-based diagnosis. The premise model-based diagnosis is that if the model itself is correct, then any discrepancies between the system's predictions and its observations arise from -- and can be traced back to -- defects in the device itself. The authors in [Hamscher et al.], however, argue, "the assumption that the model is correct is in fact necessarily wrong in all cases." Simply because it is a model, and not the actual device, it is an approximation. There will always be ways in which the model is either incorrect (contains errors in what does model) or incomplete (fails to model some aspect of the device).

As a result, applications where the device in question involves interactions that are too complicated or too subtle to be predicted with current modeling techniques may not be appropriate domains in which to apply model-based reasoning when implementing a diagnostic system.

2.9 Examples of Model-Based Diagnosis

This section describes several model-based diagnostic systems. Also included are several major works in modeling, simulation, etc. These additional works, while not complete

diagnostic systems, provide important diagnostic tools or theoretical foundations upon which much of the current model-based diagnostic research is built.

- **DART**

DART [Genesereth, 1984] was one of the systems to use deep functional knowledge in the design of expert systems. DART was a prototype for a diagnostic system in the domain of digital circuits. It uses functional models of components instead of diagnostic rules to diagnose a problem.

Both the representation language and the interface utilized by DART were relatively device-independent. Predicate calculus was used to encode design descriptions of devices under diagnosis, and resolution residue (a form of theorem proving) was used generate both sets of suspect components and test to confirm or refute fault hypotheses.

DART's problem solving was formulated using several simplifying assumptions:

1. Connections in the device are assumed to be working properly, so it seeks faulty components to account for observed fault symptoms.
2. Faults are non-intermittent -- components behave consistently for the duration of the diagnosis.
3. There is only a single fault in the device.

- **GDE**

GDE [de Kleer & Williams, 1987] retains the assumptions #1 and #2 from DART, but attempts to relax assumption #3. The challenge in diagnosing multiple faults is that the hypothesis space grows exponentially with the number of faults, as the system must now consider sets of faults rather than individual faults. GDE addresses the complexity of this problem with a combination of assumption-based truth maintenance and probabilistic inference. Diagnoses are generated in GDE using a kind of constraint propagation.

- **Diagnosis From First Principles**

Compared to DART and GDE, Reiter's theory of diagnosis from first principles [Reiter, 1987] is a more general approach. It requires only a general-purpose theorem prover to formulate hypotheses.

Reiter addresses the overall diagnostic problem, which he states as:

Suppose one is given a description of a system, together with an observation of the system's behavior which conflicts with the way the system is meant to behave. The diagnostic problem is to determine those components of the system which, when assumed to be functioning abnormally, will explain the discrepancy between the observed and correct system behavior.

Reiter's theory requires only that the system be described in some suitable logic. The theory involves working from a system description (SD) featuring a finite set of system components, and a set of observations (OBS). Both SD and OBS are finite sets of

sentences in first-order predicate logic (which includes both functions and identity). [Jackson, 1990]

Reiter sites the following as being the contributions of the theory:

1. The definition of the concept of diagnosis, including multiple fault diagnoses, based upon the preservation of the consistency of the system description and its observation.
2. The explicit use of the AB predicate for representing faults and possible relationships between faults.
3. The ability of the theory to accommodate a wide variety of logic.
4. The algorithm DIAGNOSE for computing all diagnoses.
5. Characterizations of single fault diagnoses and their computation.
6. Various results about the effects of system measurements on diagnoses.
7. The non-monotonic character of diagnosis, specifically its relationship to default logic.

2.10 Cybernet's Approach to a Model-Based System

In order to create the model, instructional and student *input* must be supplied. The model will then contain the input information, in addition to other information it computes from the input. We will first discuss the needed input and then proceed to a discussion of how this input is expanded into the actual full-blown model.

The first step in our Phase II process is to create a library of simple helicopter tasks that can be easily combined to generate more complex tasks. We will coordinate our model generation efforts with a registered helicopter instructor in order to ensure accuracy. For example we would create a simple component called 'fly straight'. Inside the 'fly straight' component would be other components such as the collective, cyclic, and rotor pedals. Values would be assigned to each component, and an adaptive decision tree would be created within 'fly straight'. Several other components such as 'bank left', 'hover in place', etc... would be created to form a basic library that any instructor can use to form more complex actions. This would turn the instruction model creation into a quick and easy drag and drop scenario. In Figure 7, you can see an example of one such component - in this case, the collective. Note that the feedback loop is shown in gray.

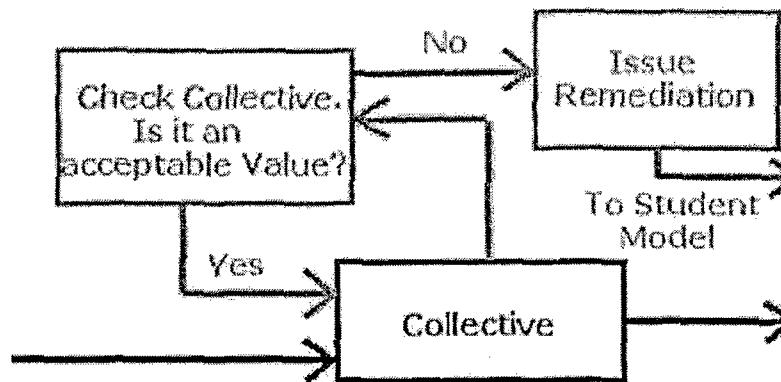


Figure 3: An Adaptive Feedback Collective Component

After the instructional model is complete, we will generate the student model. Currently, we plan on having each student take a Productivity Environmental Preference Survey (PEPS) before training. The PEPS is a 100 item self-report questionnaire that identifies individual adult preferences for conditions in a working and/or learning environment. The PEPS will give us a good idea what a student's preferred learning style is and allow us to generate an accurate student model. The student model will be composed of the student's preference for each of the 15 different learning stimuli such as *motivation*, *persistence*, *perceptual*, etc... These stimuli are covered in depth in the learning style section. Primarily, the student model will affect how feedback and remediation, as well as initial instruction, are administered to the student. For instance, a student who demonstrates a visual preference will receive instruction and feedback via visual methods, whereas a student who shows an auditory preference will receive audio instruction and feedback. While this may seem trivial from the point of looking at only one preference, when all 15 stimuli are referenced in the student model, the approach to training will vary greatly. After the student model is generated, it will then be incorporated with the instructional model to create a full model that will be used to instruct the student (Note that while the instructional model will need to be created by the instructor, the student model will be automatically generated).

Once the full model is ready, it will be used by the MBRDE\OpenSkies simulator to train the student. The model will provide the simulator with the optimum way of training the student based on the student's learning style preferences. Factors such as the student's perceptual, motivation, and structure preferences are incorporated into how the instructional information is presented. More importantly, now that the instructional model has been combined with the student model, the feedback from the instructional model will be passed through the student model before being presented to the student. As the student flies the training scenario laid out by the model, he'll be given adaptive instruction by the simulator based on his performance and the learning style aspect of the model. However, this is the most complex aspect of the proposed Phase II solution.

Since the instructional model is created out of the basic flight building blocks, the building blocks need to have the adaptive feedback trees built into them. As the student progresses through each part of the instruction model, his performance will be gauged. If his performance is not within acceptable limits, remedial feedback will be issued.

3. Model-Based Reasoning Diagnostic Engine

While rule-based reasoning and set covering algorithms can provide quick and accurate diagnoses of complex systems, they are limited to known failures and generally cannot respond to unknown conditions. This limitation greatly reduces their capability, since the number of factors involved in any complex task can quickly grow beyond the bounds of an expert system.

Model-based reasoning systems use a technique that bases their diagnoses on knowledge of the actual system models and behaviors. This technique allows for the diagnoses of problems that were unanticipated when the system was developed. Since as many failures are unforeseen, providing the monitoring agents with the capability to determine the errors is critical to the operation of these complex tasks.

However, model-based systems typically do not provide anytime diagnoses, since traversing an entire complex model will require large amounts of compute time. What we present here is a system for model-based reasoning that allows for the anytime diagnosis of errors.

3.1 Anytime Algorithms

Anytime algorithms were originally implemented to solve the problem of the limitation of knowledge-based systems with time consuming algorithms and variable performance. Anytime algorithms show an increasing quality of results gradually as computation time increases. This provides a tradeoff between resource consumption and output quality. The quality of the diagnosis is defined by the depth of the analysis or the certainty. Each of these methods of obtaining quality may be developed in several different ways. For example, as an anytime algorithm progresses, it may analyze the system in greater and greater detail. It may drop deeper down into the system hierarchy as computational time increases, providing diagnoses in varying steps. Another method to obtain this faster diagnosis is to use simpler behavioral models for calculating results at each component of the total algorithm. The algorithm would then use more complex behavioral models as more compute time is provided.

Anytime algorithms are normally defined in two different methods, interruptible and a defined computational time. The interruptible method provides a more up to the second diagnosis, however it is much more difficult to implement. In this case we decided to develop an anytime algorithm using the defined computational time with a hierarchical interface. This allows us to provide for an interruptible style for a fast, less accurate diagnosis and a more accurate diagnosis as compute time is made available.

3.2 Anytime Diagnosis

Our diagnostic framework can be characterized by its use of (1) *bottom-up modeling* resulting in the creation of a hierarchical model of complex devices under diagnosis; and (2) *top-down traversal* of the model resulting in the continuous refinement of diagnoses generated.

Bottom-Up Hierarchical Modeling

Our approach to device modeling emphasizes *abstraction* of the device's components to create a hierarchical model of the device under diagnosis. This model permits diagnoses to be produced at multiple levels of detail.

Modeling Primitives. Borrowing from graph theory, the basic primitives from which our models are constructed are *components* and *interconnections*. In addition, we have added a *data* primitive to denote observable outputs generated by the device under diagnosis.

Initial Representation. Using these primitives, a device is first modeled at its lowest level of abstraction, or its greatest amount of detail. After the components and interconnections are established, the data points are added which connect observable outputs to their origins in the device.

Data Origins. The interconnections from which data values originate are noted and entered into the device's database. These interconnections will be used to build, and also to prune, the *diagnostic tree* that is created as the device model is traversed. Once the interconnections have been recorded, the data values are associated with the source components of those interconnections.

Abstraction. Once the low-level representation has been established, repeated groupings of components into successively larger super-components create models of the device at higher levels of abstraction.

Data Tracing. As groups of components are replaced by single components at higher levels in the hierarchical model, data locations are passed up the hierarchy. The end result is that for each data value, a list is established of its source component at every level of the hierarchical model.

Example, Pt. 1

This bottom-up modeling process is illustrated in the following example. Figure 4 shows the initial representation of a device. At its level of greatest detail, this device consists of twelve (12) components, eighteen (18) interconnections, and five (5) data points.

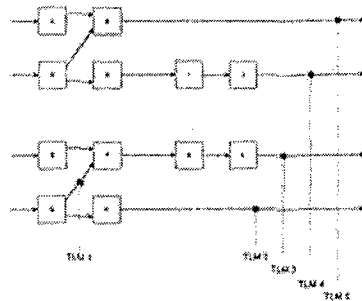


Figure 4: Initial Representation

(G, TLM1, F)
 (H, TLM2, ~)
 (L, TLM3, ~)
 (J, TLM4, ~)
 (B, TLM5, ~)

Figure 5: Data Origins

Figure 5 lists the data origins derived from the initial representations.

Figure 6 shows the initial data associations. These associations link the data values with the components from which they are output.

Figure 7 enumerates the data associations after the initial association step.

Figure 8 shows the first grouping of the abstraction process. The components are grouped to create three (3) super-components consisting of four (4) sub-components each.

TLM1: G
 TLM2: H
 TLM3: L
 TLM4: J
 TLM5: B

Figure 6: Initial Data Association Lists

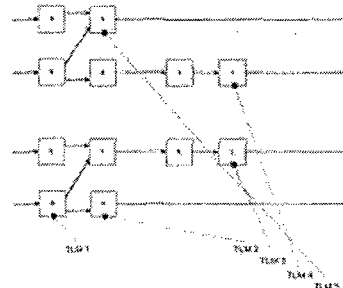


Figure 7: Initial Data-Component Associations

As the abstraction process proceeds, a component hierarchy is constructed in a bottom-up manner. The lowest level of this hierarchy is shown in Figure 9.

As this first grouping is made, the data-component associations are propagated up the hierarchy, such that each data value is now associated with its source component at this newly created level of this hierarchy. The results of this propagation of associations are depicted in Figure 10.

Figure 9 enumerates the resulting data association lists as they are kept in the device's description database.

TLM1: G, N
 TLM2: H, N
 TLM3: L, O
 TLM4: J, O
 TLM5: B, M

Figure 8: Updated Data Association Lists

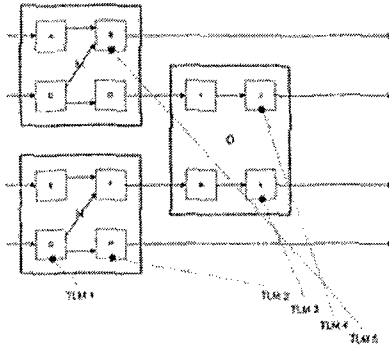


Figure 10: First Component Grouping

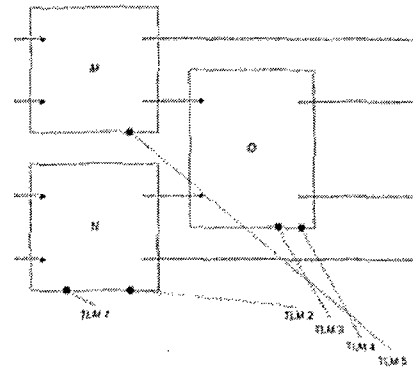


Figure 9: Second Data-Component Associations

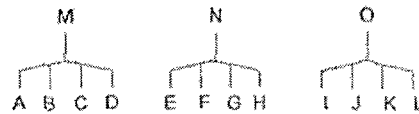


Figure 11: Initial Component Hierarchy

The first iteration through the bottom-up abstraction modeling sequence is now complete. This process ceases when a single-component level has been created as the top level in the device model, as shown in Figure 12.

TLM1: G, N, P
 TLM2: H, N, P
 TLM3: L, O, P
 TLM4: J, O, P
 TLM5: B, M, P

Figure 12: Final Data Association Lists

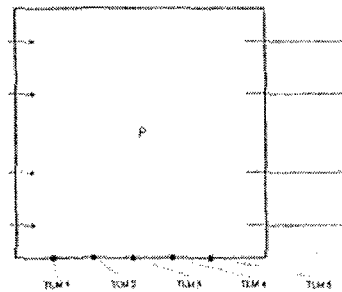


Figure 13: Final Representation

Figure 12 gives the final data association lists, corresponding to the final top-level representation. Each data value is now associated with its source component at each level in the model hierarchy.

Finally, Figure 14 depicts the final component hierarchy resulting from the bottom-up modeling process.

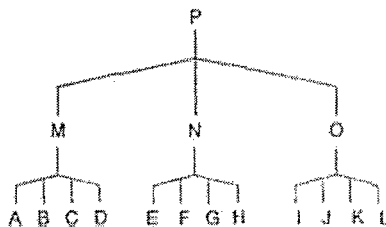


Figure 14: Final Component Hierarchy

Top-Down Diagnostic Refinement

The second half of our anytime diagnostic framework consists of an algorithm for traversing the model with the goal of deriving a tree of components making up a list of potential diagnoses for any given data anomaly. This model traversal algorithm provides an immediate high-level list of possible components in which an error could lead to the data anomaly in question, and also allows that list of components to be refined continually as time permits. Key elements of this process follow.

Propagation. The first step in deriving a tree of components that each represent potential diagnoses is to propagate a fault marker from the anomalous data value back through the model. This propagation is repeated on a sub-component basis as the component tree is expanded.

Replacement. Components in the diagnostic tree are replaced with their set of sub-components.

Expansion. Sets of sub-components are expanded to form a more detailed diagnostic tree. The fault markers are propagated through the set of sub-components to determine their proper order in the diagnostic tree, which may result in the insertion of a branch into □.

Disconnection. As a set of sub-components is expanded, an existing branch in the diagnostic tree may no longer remain connected. As the more detailed propagation takes place, fault markers will not necessarily be passed to all paths of a branch. Those branches to which a marker is not passed will become disconnected from the diagnostic tree, and will therefore be removed from further consideration.

Pruning. The diagnostic tree is pruned using nominal data observations. As fault markers are passed from component to component along their interconnections, the propagation ceases when an interconnection has been noted in the device database as the

origin of a data value and that value has been observed to be within its nominal operating range.

Example, Pt. 2

This top-down process of building a tree of potential diagnoses is illustrated in the following example, which uses the device previously modeled.

Given the observation of an anomaly at data value TLM3 with all others OK, the initial propagation through the highest level of the hierarchy is shown in steps (a), (b), and (c). Note that "~" is used to denote a path termination.

- (a) TLM3
- (b) TLM3 → P
- (c) TLM3 → P → ~

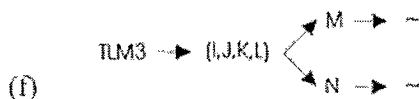
In step (d) component P is replaced with its sub-component group (O,N,M).

- (d) TLM3 → (O,N,M) → ~

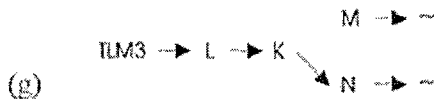
In step (e) the fault marker is propagated through group (O,N,M). The group is expanded, and a branch is added to the diagnostic tree.



In step (f) component O is replaced with its sub-component group (I,J,K,L).



In step (g) the fault marker is propagated through group (I,J,K,L). The group is expanded, and the diagnostic path connects to component N but not to component M.



In step (h) component M and anything that follows it in the diagnostic path are disconnected, as they could not be the cause of a data anomaly at TLM3.

- (h) TLM3 → L → K → N → ~

In step (i) component N is replaced with its sub-component group (E,F,G,H).

(i) $TLM3 \rightarrow L \rightarrow K \rightarrow (E,F,G,H) \rightarrow \sim$

In step (j) the fault marker is propagated through group (E,F,G,H). The group is expanded, and a branch is added to the diagnostic tree.

(j) $TLM3 \rightarrow L \rightarrow K \rightarrow F \begin{cases} \rightarrow E \rightarrow \sim \\ \rightarrow G \rightarrow \sim \end{cases}$

Finally, the path from component F to component G is pruned from the diagnostic tree. Recall from our previous model-construction example that data value TLM1 originates on the interconnection $G \rightarrow F$ and has component G as its lowest-level source. Since TLM1 has been observed to be within its nominal operating range and no other anomalous data observations have component G as their origin, fault marker propagation will not pass from F to G. The final diagnostic tree, consisting of components that are potential diagnoses for the observed anomaly at TLM3, is shown as step (k).

(k) $TLM3 \rightarrow L \rightarrow K \rightarrow F \rightarrow E \rightarrow \sim$

3.3 Hierarchical Analysis

The internal model representation builds a tree that is used to represent the student at different levels of abstraction. The root node represents the student as a unit, while the successive levels are increasingly more detailed views of the student. For example, if the tree consists of the root R with two children A and B, then at abstraction level 1 the student is simply R, while at abstraction level 2, the student is the set of components consisting of A and B.

At each abstraction level there is a graph that represents the logical dependencies of the components at that level. The program starts at the top level and proceeds until the bottom level is reached or time has run out (other alternatives are possible, such as starting at a certain level of abstraction, or skipping some levels, though the concept remains essentially the same). At each level, the program constructs a *Possible Cause List* for each of the bad data points by traversing the graph at that level. The Possible Cause List for a bad data point is a list of components such that if any one of them is defective, it could lead to the bad data under consideration. *Possible Effect Lists* are also constructed. For a given component, its Possible Effect List is the list of bad data that it could be effecting. Then the program develops a number of hypotheses that account for all the bad data, and assigns probabilities to the hypotheses. A hypothesis is in fact just an appropriately chosen set of components. Note that Possible Cause lists refer to one particular data point, while a hypothesis takes all the bad data points into consideration. Details concerning what makes a set of components a hypothesis will be discussed later.

Note that this algorithm proceeds in an "anytime" fashion. The computations are done at each level, thus providing a complete, though possibly vague, diagnosis at each level; in addition, the computations at successively deeper levels provide increasingly more detailed diagnoses. The program essentially has two parts. In the first part an appropriate model is constructed from the given information. The second part uses this model to draw conclusions. The first part is done prior to any serious concern over time efficiency, while the second part is expected to be run in real time.

3.4 Internal Model Construction

In order to create the model (the first part of the algorithm), the algorithm needs to be given particular information, the *input*. The model will then contain the input information, in addition to other information it computes from the input. We will first discuss the needed input and then proceed to a discussion of how this input is expanded into the actual full-blown model.

3.5 The Input

The input (and the model for that matter), can be broken into two aspects, a single hierarchical tree and a set of dependency graphs. These two aspects are *conceptually* independent, but *physically* dependent. By this we mean that graphs and the tree can be defined and conceptualized without the other, but that the actual pieces they contain are shared; in particular, some nodes can be shared, constituting what is a physical-like (of course it's not really physical!) dependency. However it is probably simplest to understand the tree as existing first, with the graphs built around the tree; so in this sense the tree is independent of the graphs, but the graphs depend on the tree. The word "node" and "vertex" will be used to refer to the same entity, though the former when speaking of a tree and the latter when speaking of a graph.

3.6 The Tree

We now further develop the idea of how the tree is used. As mentioned in section 4.2, the tree is used to represent the student at different levels of abstraction. If node P has exactly the children A, B, and C, this indicates that P is a component that is *made up of* the three components A, B, and C. *To say P is made up of A, B, and C means that P contains these 3 nodes and P contains nothing more than these 3 nodes;* thus P is no more than another name for the set containing A, B, and C.

Note that the tree does not necessarily represent something physical. The tree provides a hierarchical categorization of the physical student. Consider an example tree T in which the node P has children A, B, and C. It could be the case that the components represented by A, B, and C are physically contained in a box and so P could be seen as corresponding to something physical, namely, the box and its contents. However, it could be the case that A, B and C are simply 3 different components with no apparent physical relationship. Yet there could be a reason to categorize these 3 components into a single super-component P. In this case, though A,B, and C could be physical, P is not; rather, P is a category that includes 3 physical objects A, B, and C. In general, we would expect the

leaf nodes to be physical objects and the non-leaf nodes to be *abstractions* or *categorizations*.

Each component of the student has a probability of being defective. Actually only the probabilities for the leaf nodes are needed and from these probabilities the probabilities of the non-leaf nodes can be computed.

3.7 The Set of Graphs

The tree introduces a number of nodes, indicating how these nodes relate to one another as far as containment. The set of graphs then expands upon this to indicate how the components depend upon each other in terms of some other characteristic. These nodes are the point of physical dependence between the tree and the set of graphs. We discuss nodes in more detail later.

We assume that we are given a representation of a student that indicates the components and the dependencies between the components. This representation would typically be a schematic diagram of the student circuitry with additional dependencies, such as temperature, indicated. This representation should somehow (automatically by computer program or by user) be converted into a set of directed graphs S , in which the vertices of a graph represent the components of the student (already defined by the tree) and the edges of a graph represent the logical dependencies between the components.

First we give an informal explanation of this concept. The set of graphs S could for example contain two graphs VT and TP that represent the dependencies voltage and temperature, respectively. We will call voltage and temperature *dependency types*. An edge in the graph VT with source node s and target node t indicates that component s depends on component t , in terms of voltage. We will have a separate graph for each dependency type; the vertices will be the same for these graphs since we are still considering the same components, though the edges, which represent relationships, can be quite different.

Formally, the program will be given a set of dependency types (or we could call these *graph names*) D_1, \dots, D_r . It will also be given a set of graphs $A^1 = (V^1, E^1), \dots, A^r = (V^r, E^r)$ that indicate the component dependencies for each of the dependency types. *If component A depends on component B, in terms of dependency type D_i , then the graph A^i should have an edge from vertex A to vertex B.* In order for A to depend on B, B needs to be giving some kind of output to A, so *in terms of the direction of data flow (the meaning of data is being taken loosely) there is an edge in the opposite direction, from B to A.*

Some edges are called *data edges* because they have *data points* on them. More specifically, for each dependency type D_i , a number of edges will be specified as data edges of the type D_i . These edges contain data of type D_i and appear in graph A^i . If a data point T is on a data edge (A, B) this indicates that the data is output by B and then taken into A, thus the reason that A depends on B.

Any graph is allowed that meet the following two criteria:

- 1- Only data edges can have the same source and target (i.e. be *loops*).
- 2- No two nodes in a graph can be on the *same path to the root* (or be *directly related*). Two nodes are directly related if repeatedly applying the parent function to one of the nodes, eventually yields the other.

We now come to the discussion of different kinds of nodes. There are two kinds of nodes, *component nodes* and *virtual nodes*, the former being the type that is shared between the graphs and the tree. The *component nodes* represent functional objects in the student, while *virtual nodes* represent non-functional objects. However, virtual nodes actually have a purpose, albeit a small one. Virtual nodes are used to represent the intersection points of the dependencies, such as is common on a circuitry schematic, in which black dots represent intersecting wires. It could be possible to do with out virtual nodes, however this would require converting *diagrams with virtual nodes* into a model without any; there are odd scenarios in which this conversion becomes confusing.

So in summary, the program expects the following as input:

- 1- A tree T made up of component nodes C .
- 2- A set of virtual nodes VT .
- 3- A set of directed graphs $S = \{ A^1, \dots, A^r \}$ named D_1, \dots, D_r , respectively, in which the nodes of any A^i are contained in $(C \cup VT)$.
- 4- For each $A^i = (V, E)$ a set of data edges $T_i \subseteq E$
- 5- The function probability: $C \rightarrow \{\text{numbers from 0 to 1}\}$, indicating probabilities of being defective (need only be defined for leaves of T).

3.8 The Expansion of the Input to Create the Model

From the set of graphs and the tree, a collection of higher-level graphs is formed that are consistent with the given graphs, but contain less detail. The program determines how many levels are in the tree T ; call this number m . The m levels of T are numbered 1 through m from top to bottom. More specifically, the root is at level 1, and another node n is at level $2 + [\text{number of nodes between it and the root}]$. Each graph then also has a level associated with it, namely the level of the node with the largest level, that it contains. In many cases it is expected that each of the graphs will contain all the leaf nodes, so that the graph is a *complete description* of the dependency relationship at the lowest level (or physical level). The higher level graphs would then be less detailed abstractions of the low level physical description.

For a graph A at level t , the program will construct graphs $A_1, \dots, A_{[x1]}$, where graph A_k is the graph corresponding to level k of tree T . Graph A_t is set equal to A and then the program constructs $A_{(k-1)}$ from A_k , for $k = t, \dots, 2$ placing the edges so that the graph $A_{(k-1)}$ is consistent with the graph A_k . We will give the *Graph Construction Algorithm* that will make use of the following function:

Given a node n and a natural number k define

$$U(n,k) = \text{parent}(n), \text{ if } n \text{ is at level } k \\ n, \text{ otherwise}$$

3.9 Graph Construction Algorithm

The construction of graph $A_{(k-1)} = (V_{(k-1)}, E_{(k-1)})$ from $A_k = (V_k, E_k)$

1 - $V_{(k-1)} = V_k$ with all the vertices at level k in the tree T replaced by their parents

Initialize $E_{(k-1)} = \{\}$

2 - For each edge (u,v) in E_k do:-

a - $u' = U(u,k)$

$v' = U(v,k)$

b- if $u' = v'$ (i.e. the edge is a loop) AND

(u,v) is a data edge

then

$E_{(k-1)} = E_{(k-1)} \cup (u', v')$, and (u',v') has all the properties of (u,v)

if $u' \neq v'$ then

$E_{(k-1)} = E_{(k-1)} \cup (u', v')$, and (u',v') has all the properties of (u,v)

We see how the graph $A_{(k-1)}$ is consistent with the graph A_k , in that any edge (u,v) in $A_{(k-1)}$ is represented in A_k as an edge (w,z) , where w could be a child of u and z could be a child of v . Notice that the above Graph Construction Algorithm will only create loops (edges having the same nodes for its source and target) if the edge is a data edge. A crucial point is that the higher level graphs that are constructed have edges with the same properties as the ones they are derived from. This includes their name and the fact that they have the same data point on them. This also includes any changes made to one of the edges; what is really going on is that there is one edge existing on different levels.

Probabilities of being defective should be passed up the tree from the given probabilities at the leaves, as done in the following algorithm:

3.10 Probability Percolation Algorithm

Starting at the leaves and moving up to root, pass up probabilities as follows:

Given a node N with r children c_1, \dots, c_r having respective probabilities of being defective p_1, \dots, p_r then

$$\begin{aligned} \text{probability that } N \text{ is defective} &= \text{probability that at least one of } c_i \text{ is defective} \\ &= 1 - \text{probability that none of } c_i \text{ are defective} \end{aligned}$$

$$= 1 - (1-p_1)(1-p_2)\dots(1-p_i)$$

3.11 Drawing Conclusions In Real Time

The process of constructing the graphs and other work to this point is done before serious concern over time efficiency. When the data points are declared to be either good or bad, the real time process begins. In the following discussion we will use the following notation:

A_k^j \equiv the graph with dependency type D_j at level k

The discussion of the algorithm is difficult to describe sequentially because it can respond to different real-time user instructions. Instead of describing an algorithm, the basic capabilities will be discussed.

Given a level in the tree, the program can do three basic computations. The program can compute the Possible Cause Lists (P.C. lists), the Possible Effect Lists (P.E. lists), and the Minimal Covers (M.C.'s). A typical approach would be to start at level 1, compute the Possible Cause and Effect Lists for this level, then go on to level 2 and do the same, and so on. Then when the bottom level of the tree is reached or after a certain amount of time has elapsed the program will stop moving down the tree and the Minimal Covers will be computed at the lowest level reached. The Minimal Covers could be computed at each level, but since they can take a fair amount of time, they are only computed at the lowest level reached.

This scheme is only one of many possible approaches. The Minimal Covers could be computed at each level or the program could skip some levels, for example. Given that there is such variability, we will discuss what these 3 computations are and how they are arrived at and go no further for now.

3.12 Possible Cause Lists

A Possible Cause List can be computed for a bad data point at a particular level L in the tree. Recall that some edges are data edges and contain data, which can be bad or good (or unknown). The Possible Cause List for a particular bad data will be a set of component nodes; the list gets its name because if any of these component nodes are malfunctioning it could be causing the bad data under consideration. To understand precisely what they are we give an overview of the algorithm used to compute them.

Given a bad data point T , we want to find the P.C. list of T at level L .

- 1- Let E be the data edge that T is located on at level L . Recall that data points are located on data edges and the same edge exists at multiple levels, all with the same data point.
- 2- Let G be the graph that E is in at the given level.
- 3- Let P.C. be the set of all nodes in G that are reachable from E by following the edges in the standard direction from source to target, subject to one restriction:
 - If an edge contains a good data point, then the edge is not traversed.

The rationale behind this algorithm is to put a component node on the P.C. list of a bad data point, if the data point depends on that component node, thus the reason why the above algorithm calls for traversing the graph in the direction of dependence, from source to target. Now consider the reason for not traversing an edge containing a good data point. Assume the graph is being traversed and the node A has been included in the P.C. list and the edge (A, B), which contains a good data point, is in G. Since A could be causing the bad data (we know this because it is on the P.C. list) and A depends on B, it could be the case that B is defective and thus causing the bad data. However, the good data is output by B to A which indicates that though B could have problems in other areas, it is giving good output to A.

3.13 Possible Effect Lists

A P.E. list can be computed for a node at a particular level L in the tree. This is the list of data points, possibly of different dependency types, that the node could be causing to be bad; so these data points are the ones that the node *effects*. Such a list could be constructed in a way very similar to constructing P.C. lists. In this case, the algorithm starts at the node in question and traverse the edges *in the opposite direction, going from target to source*. Again, edges with good data points would not be traversed. Besides traversing in the opposite direction, another difference is the fact that a P.E. list can encompass a number of dependency types. For this reason, in the mentioned traversal, what is actually meant is a separate traversal for the graph of each dependency type at the given level. However, this computation can be accomplished without a traversal, by simply looking at the P.C. lists for all the dependency types, and computing the P.E. lists for all the nodes in the P.C. lists (nodes not in the P.C. lists will have empty P.E. lists).

- 1- Initialize the P.E. list of all the nodes at to be empty.
- 2- For each P.C. list P for data T, at level L, do the following:
 - For each node N in P, put T on the P.E. list of N.

3.14 Minimal Covers

We now discuss what a *minimal cover* is by beginning with a motivation for its definition. Given the student at some level of abstraction with its bad data we want to find hypotheses that explain all the bad data. A hypothesis is simply a set of components such that all of them being defective provides an adequate explanation of the bad data. However we also don't want to provide *too much* information in our hypothesis, thus each component in our hypothesis is important to the explanation of the bad data. We capture this idea of hypothesis formally in the following definition of a minimal cover.

Given a directed graph $G = (V, E)$ and a set of bad data edges $B \subseteq E$, we say that a set of vertices $C \subseteq V$ *covers* B if for every edge in B there is a path from some vertex in C to this edge, *moving in the opposite direction of the edges from target to source*. We go in the opposite direction of the edges since given a vertex v, moving in the direction of the edges finds components and edges *on which v depends*, while moving in the opposite direction finds edges and components *that depend upon v*. A cover is thus an adequate

explanation for the bad data. Given a cover C, we say that it is a *minimal cover* if removing any vertex from C causes C to cease being a cover. So a minimal cover is an adequate explanation of the bad data that doesn't contain *too much* information.

The computation of the minimal covers takes exponential time in the worse case, so a backtracking approach is used to reduce the time expense. The minimal covers are actually computed from the P.C. and P.E. lists, though they could be computed directly from the graphs.

3.15 Overview of Algorithm to Find Minimal Covers

The following describes how the minimal cover finding algorithm works. It complements descriptions that are included in the software modules themselves. To make referral to the software easier, references are made to the class hierarchy² in the software as indicated in the footnote.³

The sets of the algorithm sequentially are:

- 1- Let C be the empty sequence (this represents the current solution)
Let S be the empty set of solutions (will contain all the minimal covers at the end of the algorithm)
Let N be a sequence of nodes obtained from all of the possible cause lists (these are the nodes which can be in a minimal cove, this is a static entity).
- 2- If there is a next node from N to add to the current C, then add it,
otherwise, remove the last element added and add next node (IF POSSIBLE).

2 Specifically, refer to the description of SearchTree class and especially the method GetNextSolutionByBacktracking. Also refer to the MinimalCover class and especially the methods AddElement and RemoveLastElement. The precise description of the algorithm is in essence contained in these places. *There are explanations in these classes that complement this overview document well. Also, the references in SearchTree are valuable, and the ideas are similar.*

3 Classes involved with Minimal Cover Computation include the following, with class hierarchy indicated:

- 1- SearchTree
- 2- UnorderedSolutionSearchTree: SearchTree
- 3- MinimalCoverSearchTree: UnorderedSearchTree
- 4- SolutionContainer
- 5- Cover: SolutionContainer
- 6- MinimalCover: Cover

- 3- If this last remove/add is not possible then we are done and S contains our minimal covers.⁴
- 4- If C is a minimal cover then add C to S
 If C is "Bad" (see below) or C is minimal cover then backtrack, meaning that we remove the last element added to C⁵, and thus cut off the possibility of any future extensions in this direction.
 Otherwise, we have a C that is not yet a minimal cover, but which could conceivably be extended to one.
- 5- Goto (2)

Further Comments on above:

C is "Bad" if it is not a minimal cover, and furthermore could not possibly be extended to a minimal cover by adding nodes. As soon as we know C is Bad, we stop extending in that direction. Badness is tested for by the algorithm in MinimalCover::isMinimal (Note that this returns false if "Bad" and true not "Bad"). What happens is that each node is examined and each should uniquely cover some data, otherwise it could be removed with no effect, and so the cover could not possibly be minimal.

Illustrative Example of the Algorithm:

To illustrate the algorithm conceptually, we use the following example graph:

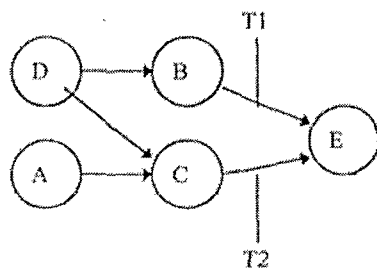


Figure 15: Example Graph (for Illustrating the Minimal Cover Computational Algorithm)

Let G be the following graph (shown to the left):

Vertices = a,b,c,d,e

Edges(going from source to target) = (d,b),(b,e),(d,c),(c,e),(a,c)

Data = T1 on (b,e), T2 on (c,e)

We have possible causes (P.C.) lists

P.C. of T1 = [d, b];

P.C. of T2 = [a, c, d]

⁴ via MinimalCover::AddElement and MinimalCover::RemoveElement

⁵ via MinimalCover::RemoveLastElementAdded

The minimal covers we should find are:

$\{d\}, \{a, b\}, \{b, c\}$

The following is an illustrative run of the algorithm, with comments:

- 1- $C = \langle \rangle$, empty sequence, $S = \{\}$, $N = \{a, b, c, d\}$ ⁶
Put C in the root of a tree⁷
 - 2- $C = \langle a \rangle$. Put this as the leftmost child of C (or $\langle a \dots \rangle$), the root.
 - 3- C only covers T2 so it is not a cover. But it is not Bad either, so goto (2).
 - 2- $C = \langle a, b \rangle$. Put b as child of $\langle a \rangle$
 - 3- C is a minimal cover so make $S = \{ \langle a, b \rangle \}$.
Remove last element added, namely b. So we go up tree to node $\langle a \rangle$.
 - 2- Add c, so $C = \langle a, c \rangle$
 - 3- C is Bad since both a and c ONLY cover T2, and so neither one has data that it uniquely covers. Actually we only needed one such bad node. No matter what is added to $\langle a, c \rangle$ it could not possibly be minimal because either a or c could always be removed.
So we remove last element added to get $C = \langle a \rangle$; go back up tree.
 - 2- Add d, so $C = \langle a, d \rangle$, rightmost child of $\langle a \rangle$ in the search tree.
 - 3- C is bad. In this case d is actually a minimal ON its OWN.
Remove d to get $C = \langle a \rangle$. Nothing is left to add to C so remove the last element added, a, to get $C = \langle \rangle$; and we are back at the root of tree.
- The following is a summary of what will happen from this point in the process forward depicted by the variable changes (at every step):
- $C = \langle b \rangle$,
 $C = \langle b, c \rangle$ - is a minimal cover
 $S = \{ \langle a, b \rangle, \langle b, c \rangle \}$
 $C = \langle b \rangle$
 $C = \langle b, d \rangle$ - is Bad

⁶ This set does not include "e".

⁷ Such a tree is implicit in the running of the algorithm, and is essentially a search tree, with possible solutions at its nodes.

$C = \langle b \rangle$

$C = \langle \rangle$

$C = \langle c \rangle$

$C = \langle c, d \rangle$ - is Bad

$C = \langle c \rangle$

$C = \langle \rangle$

$C = \langle d \rangle$ - is a minimal cover

$S = \{ \langle a, b \rangle, \langle b, c \rangle, \langle d \rangle \}$

$C = \langle \rangle$

There is nothing else to add, so the algorithm is done

Limitations of the Current Algorithm:

A possible problem in the algorithm that occurred to us after testing and debugging, is that if a model has a cycle in its graph so that a node "a" could actually depend on itself problems in resolution might occur.⁸ Perhaps later we should check with examples, and if this is a real problem we can patch it up either by disallowing such models to be constructed, or allowing them and altering the algorithms slightly. The first approach may be easier and actually makes more sense. This makes for a better system by being appropriately inflexible, unless there could be a use in the future for such constructions.

4. Learning Styles

One of the main factors that must be taken into account when teaching students is their preferred learning style. Everyone has a distinct learning style. Some styles are more similar than others, but students have their own set of conditions that if met, provide them with their ideal learning environment. However, most computer simulators often have the tendency to teach using only one style. While this may be perfect for those students who learn best through that style, those that don't will experience difficulties learning and have a much harder time grasping the required concepts. Kiernan (1979) stated:

We now see that part of the problem was the tendency to apply a single (instructional) approach to all students... Student learning style challenges this premise and argues for an eclectic instructional program, one based upon a variety of techniques and structures, reflecting the different ways that individual students acquire knowledge and skills. (p.i)

⁸ This is nonsense because not real system would have a fault which depends on itself, but it is theoretically possible to code such a model up.

Simulator software that provides a varied approach to learning, and that matches instructional methods with each student's style preference, can result in improved attitudes toward learning and an increase in productivity and achievement.

4.1 Diagnosing a Student's Learning Style

The most comprehensive way of determining an adult student's learning style is through the Productivity Environmental Preference Survey (PEPS). The PEPS is a 100 item self-report questionnaire that identifies individual adult preferences for conditions in a working and/or learning environment. The PEPS contains: (1) measurement of 20 elements on a five-point Likert scale, (2) development by content and factor analysis, and (3) reliability data equal to or greater than .60 for 68% of the 20 elements. By administering a PEPS beforehand, the simulator can create a model of the student's learning style and then choose an appropriate initial instruction strategy. The student, instructor, or simulator can then modify this strategy as training progresses. In the following sections, each of the different learning style stimuli that can be addressed by simulation software will be discussed, along with a method of implementation.

4.2 Physical Stimuli

Physical stimuli are aspects of the student's physical state that provides the student with his optimal learning environment. The four main categories of physical stimuli are perceptual, intake, time, and mobility. Only one of these factors, perceptual, can be taken into account in the simulator software. The instructor should regulate factors such as intake (whether a student likes to eat while he learns) or time (what time of day that student likes to learn) instead of the simulator. However, simulation software can address the student's perceptual preference, which will be described below.

4.3 Perceptual

Perceptual stimuli are the ways in which we acquire knowledge. The three classes of perceptual stimuli are tactual-kinesthetic, auditory, and visual. A number of studies verify that students' learning is enhanced when they are taught through their personal perceptual preferences (Urbschat, 1977; Carbo, 1980; Weinberg, 1983; Wheeler, 1983; Jaronsbeck, 1984; Kroon, 1985; Martini, 1986).

Tactual-kinesthetic (TK) learners are students who prefer to take a "hands on" approach. They learn through doing. The best way for them to learn any task is through firsthand experience. An adaptive helicopter simulator that allows a TK student to immediately start the simulation, and then provide auditory or visual prompting as he flies, will provide this student with a better learning environment. For example, an introductory TK lesson could start with the helicopter at 1,000 feet. Pre-recorded audio instructions explaining to the student that he or she must maintain a steady heading through the use of the cyclic and tail rotor pedals would be broadcast over the headphones. As the student tries to maintain a steady heading, the simulator would prompt the student to apply more pressure to the right or left rotor pedal, pull back on the cyclic, or make any adjustment to the controls that were necessary. If the student is not able to maintain the course, the simulator would re-center the helicopter and have the student try again.

Visual learners learn best by reading information, or viewing a slide/movie presentation. Any form of media by which the student can see the material to be learned will be effective. A visual student would gain the most out of reading an instruction manual or textbook before he begins training. An adaptive helicopter simulator that begins by presenting the student with written material or visual presentations and then allows a student to reference this material during the lesson would provide an effective approach. The student would also benefit from being able to access written instructions and to receive visual prompts during the simulation. For example, given the same scenario as the TK learner above, written descriptions of how to use the cyclic and rotor foot pedals would be displayed on screen, followed by a description of the introductory lesson. As soon as the simulation began, visual prompting would be administered to correct the student. A picture of a red left rotor pedal popping up in the lower left corner, signifying that the trainee should push it, would be an example of a visual prompt.

Auditory learners are likened to having a tape recorder inside their head. They are usually able to remember conversations well, and learn the best from lectures, and other auditory stimuli. The best way for this student to learn is by listening to pre-recorded information, and responding to it in a verbal manner. An adaptive simulator for an auditory learner would give the student audio instructions with visual aids. It would also allow the student to use voice recognition software to "talk back" to the simulator via microphone in order to replay old information or obtain more detailed information on a given subject. Given the same example as the TK and visual learners, a student would be given verbal directions as if there were an instructor in the co-pilot's seat. As the student piloted the helicopter, the simulator would give the student advice like "Apply a little more pressure to the left rotor pedal" followed by "That's a little too much" if they pushed on it too hard. The student would also be able to query the simulator with verbal commands like "Repeat instructions" or "Rotor pedal description."

As long as students can be addressed primarily in their preferred perceptual learning style, you can combine aspects of each style to effectively give the student a wider range of options and more complete training. For instance, a TK learner may find auditory feedback like "Move the cyclic to the left" useful, while an auditory learner might like to look at diagrams or presentations once in a while, or a visual learner be able to use voice recognition commands to view written material. The idea of developing a model of the student's perceptual preference is to give the simulator an idea of how to start the training. Perceptual preferences will not be used to limit the student, and all options that are available to the other preferences can be accessed and changed through a readily accessible user menu.

4.4 Environmental Stimuli

The only environmental stimuli that can currently be addressed in a flight simulator, without making extensive changes to current hardware, is sound. While it is true that all of the environmental stimuli could be addressed (Temperature, Light level, Design, and

Sound), it is impractical, expensive, and sometimes counterproductive to have thermostats, dimmer switches, and reclining seats built into military training simulators. Therefore they will not be covered in this paper. The only environmental stimuli that can be addressed without significantly altering the purpose and cost of training is sound.

Sound. Most flight simulators address sound in one of two ways, there is either an absence of sound during instruction, or there is a mixture of background music and/or sound effects. However, this approach fails to address the individual sound preferences of each student. Some students prefer to learn with sound, while others do not. Schmeck and Lockhart (1983) suggest that inherited differences in nervous system functioning require that extroverted individuals learn in a stimulating environment, while introverted persons prefer a quiet, calm environment with few distractions. In addition, Pizo (1981) found that when sixth grade students were matched with their preferred acoustic environments and the presence or absence of sound, these students scored significantly higher in reading achievement and evidenced more positive attitudes toward school than students who were mismatched on this element. A simulator that addresses sound would start training according to each student's preferred audio environment and also provide the student with the choice to toggle sound effects and background music on or off in addition to volume control.

5. OpenSkies Virtual Environment Training System

The OpenSkies Training system provides an interactive development system to train both students and instructors in a Virtual Reality Environment. Many simulators have the capability to familiarize the student with simulations of the actual instruments and a few have the capability to create scripts for mission play. OpenSkies is the only one to have so closely integrated the analysis and performance measurement capabilities directly within the simulation software.

5.1 Why is it needed?

OpenSkies is based on the training methodologies developed in Naval Research Labs for actual Navy training. OpenSkies was created with the following key ideas behind it.

- To improve the student's performance beyond current training program capabilities.
- To provide a measurable performance standard.
- To provide the fast, simple creation of new training courses.
- To provide a low cost solution to training on complex, expensive equipment.

5.2 How will this system improve the student's performance?

This system will draw better performance out of the student by:

- Providing for a more quantitative approach to performance measurement.
- Providing a more structured environment in which instructors may teach.
- Showing students exactly where their deficiencies exist and allowing them to concentrate on those items.
- Ensuring that the student meets a specific level of overall performance or performance in particular areas.

5.3 What are the advantages of this system?

- Low cost - requires only sub-\$1000 PC.
- Networked multi-participant capabilities for team training.
- Applies event-based training methodologies to a virtual environment.
- Allows for recording and playback of the entire training mission for later analysis.
- Provides automatic performance analysis feedback to the student.

5.4 What is unique about this system?

- Applies a quantitative approach that allows for a better comparison of performance.
- Instructors may 'ride' along for real-time instructor analysis.
- Provides for tracking of class level of performance as well as instructor level of teaching.
- Provides for the development of training scenarios in hours rather than days or weeks.
- This system may be customized to any domain for faster scenario development.

5.5 The OpenSkies Scenario Development and Performance Measurement

The OpenSkies interface contains tools for easily creating new scenarios and directly testing the student on the training exercises. The student's exercise is completely recorded for later analysis by the instructor as well as automatically analyzed to determine which objectives the student has or has not met in each exercise. This interface includes:

- point and click programmable scenario
- dialog boxes to test the student during the scenario
- initial conditions of the actual scenario

- the capability to track all of the student's responses, or lack thereof, to any event
- collection and playback capability for any exercise
- instructor analysis of student performance
- automated performance analysis
- Additionally, the system contains a complete application-programming interface for adding complex and new event types.

Employing these interfaces, OpenSkies is able to produce a virtual environment training system that provides realistic training and a complete performance analysis package for training both students and instructors.

5.6 Scenario Development

This capability provides the instructor with a simple interface for quickly developing new courses.

Scenario Purpose and Objectives

This interface allows the instructor to define the scenario objectives for the student through a point and click dialog box. These objectives are entered into the objective database by the instructor and are generally defined for the specific type of training.

Object Initialization

This interface provides a dialog box to set any initial conditions of the students vehicle/avatar interface. Examples include limited amounts of fuel for vehicles or an unfamiliar tool set for the avatar.

Scenario Events

This dialog box allows the instructor to select particular events to solicit responses from the student. This can be interactions with other objects, test questions, etc. This interface can be configured to be domain specific. For example, a flight training domain would set up pre-flight, takeoff, en route, etc. sections for development and focus on communications, navigation and situational awareness skills for training the student.

Briefing/ Debriefing

This interface provides the instructor with the capability to brief/debrief the student. This consists of a text and MIME interface for providing information. The MIME interface allows the instructor to attach any type of document to the briefing including such items as video, audio, word documents, HTML documents or any other information accessible on the computer system.

Environment

This interface allows the instructor to change the weather and other environmental conditions of the scenario.

5.7 Performance Analysis

This capability is provided by several pieces, recording and playback, marking of events and mission evaluation and mission summary.

Recording and Playback

The student's mission is automatically recorded for later playback and analysis. This includes all interactions with the system, such as the simulation environment, popup questions and responses.

Marking of Events

The instructor may mark events in the system while monitoring it in real-time, or playback for later analysis of particular events.

Mission Evaluation

This interface uses the recording and playback capabilities of the system. It allows the instructor to playback the student's scenario. The instructor may jump ahead to particular events and play them, as well as rate the student's performance for any particular event or objective.

Mission Summary

This summary effectively scores the student on his/her performance. The system logs all events and objectives that the student did or not make and gives the student points for successful objectives.

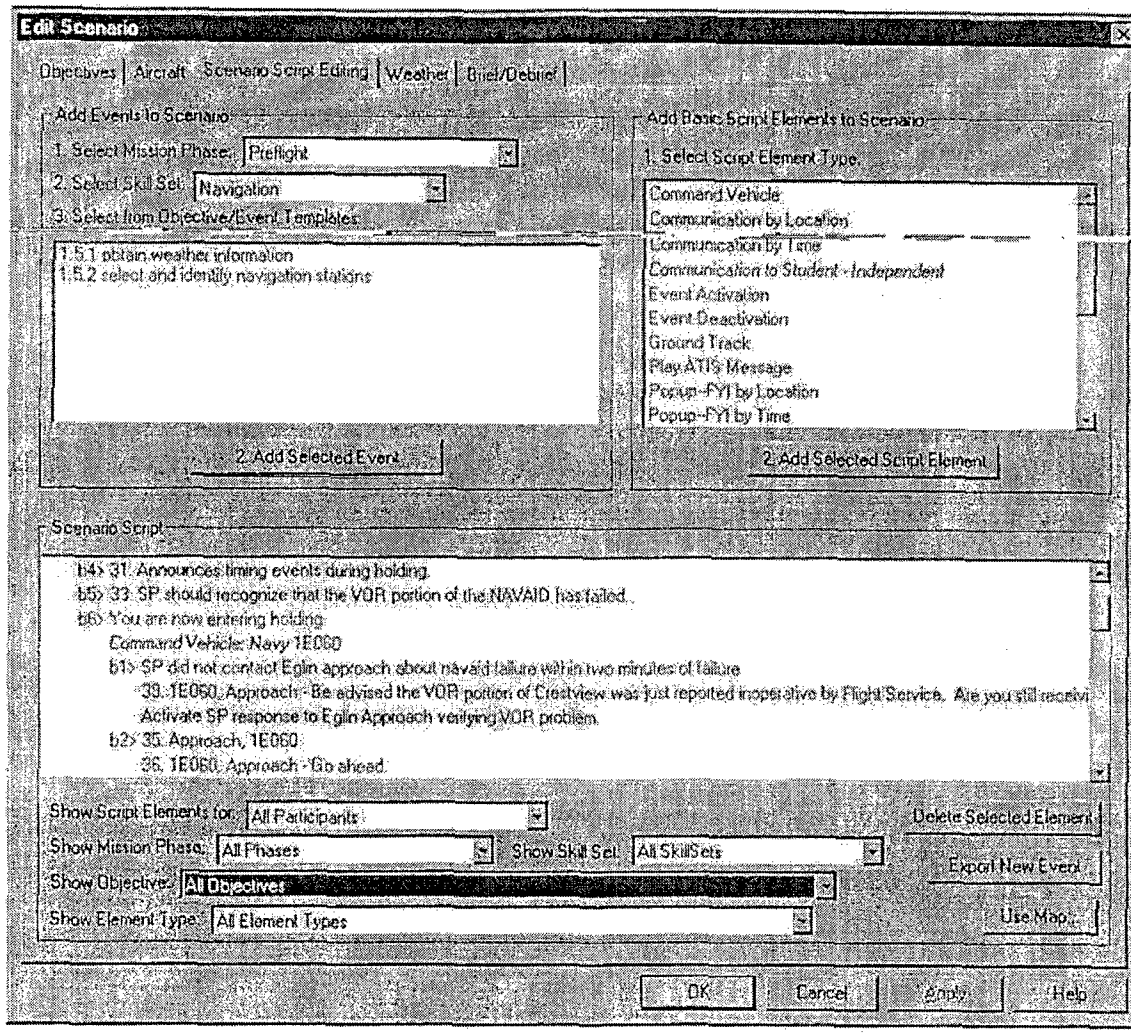


Figure 16: Scenario Edit Window

5.8 Example Scenario Development

We present here a simple example scenario development for a pilot doing a pre-flight checkout. Assuming we have already developed the domain specific objectives and skill sets, the instructor can quickly develop a new scenario. The instructor starts by entering any data about the aircraft type and initial state, such as the amount of fuel it has. The instructor enters the main section of the scenario development, the Scenario Events interface, where he/she is able to create the actual script content.

The instructor develops the script content in the window shown in Figure 19 via the following steps:

- 1) The instructor starts by selecting the initial 'Phase of the Mission' for the course. For this particular domain, this would be 'Pre-Flight'.
- 2) The instructor selects the skill set that the student will be trained on. The instructor may select from such items as 'Instrument Checkout', 'Request Clearance', or 'Taxi to Runway'. Assuming the instructor selects 'Request Clearance', the interface provides a list of events available for these criteria for addition to the scenario.
- 3) The event choices will now be limited to a few items such as 'Tower Communication', or 'Radio Traffic', or 'Change Frequency', representing the events that may occur at this point in the scenario. For this example, the instructor selects 'Tower Communication' and adds it to the scenario. The interface presents the instructor with a list of sub-tasks that will occur.
 1. The student initiates a call to the tower requesting clearance.
 2. The tower acknowledges the call from the student and requests standby.
 3. The tower provides clearance to the student.
 4. The student acknowledges the clearance to the tower.
 5. The instructor selects each of these sub-tasks and defines the specific inputs and outputs of these sub-tasks. The following dialog defines inputs for the clearance from the tower event.

Instruction for Student by Time

Objective: ☒

Radio Channel:

Time Offset:

☐ From Beginning of Scenario
☒ From Another Event

Record Audio

Message:
1. Navy 1E060, Pensacola Approach - Proceed direct Crestview, contact Eglin Approach on one two four point zero five.

These include:

- the objective,
- the radio station of the student,
- time offset from the beginning of the scenario or from another event,
- the maximum score that the student may achieve for this event,
- the recording of the audio from the external source such as the control tower,
- a description of the actual message.

Figure 17: Instruction for Student by Time

5) Once the instructor has finished entering the events and their inputs, the scenario is saved.

6) The instructor may then test the scenario by executing the application.

This system allows the instructor to quickly develop the course based on specific skills for each particular domain. Further, by defining the domain prior to the course development, the instructor can easily understand and develop the course in a familiar manner.

5.9 Running the Example Scenario

The interface for executing any scenario is simple and straightforward.

- 1) The student logs in and loads up the particular scenario.
- 2) The student is then presented with the briefing for the scenario and starts the exercise in the aircraft with the initial conditions set by the instructor. This may include the flight path of the scenario if the instructor wishes.
- 3) The student then executes the scenario and responds to events such as popup dialogs for situational awareness, radio calls from the tower and any other events programmed by the instructor. The system records the entire scenario, watching for responses to all events.

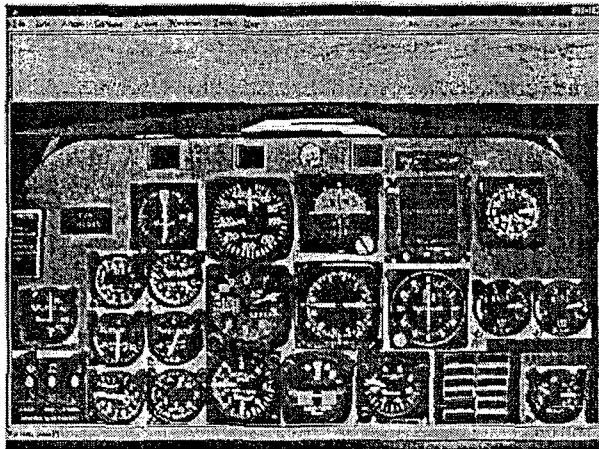
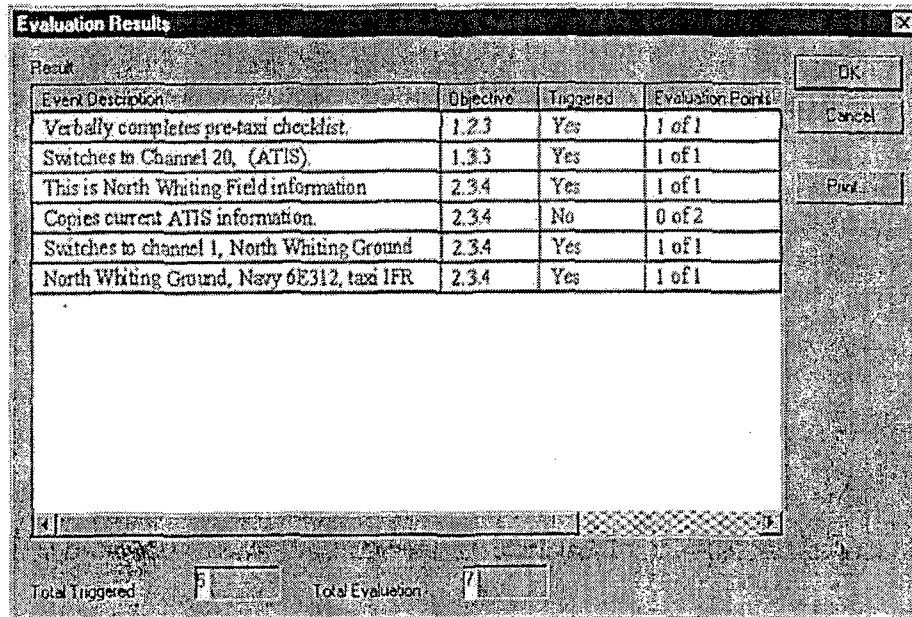


Figure 18: Control Panel

- 4) Once the student has completed the scenario, he/she is presented with the summary analysis of his/her performance. This provides the student with a rating and analysis of which objectives were met.
- 5) The student may then go back and try the scenario again to correct any errors that occurred in the first run.

This interface is all automatic, running the student through the scenario without the student needing any training on the software prior to executing the scenario. This provides a simple interface that a student may use at any time. Also, since the system runs on desktop PCs, the student may actually practice at home on a desktop PC.



The screenshot shows a window titled "Evaluation Results" with a table of event performance. The table has four columns: "Event Description", "Objective", "Triggered", and "Evaluation Points". There are six rows of data. To the right of the table are three buttons: "OK", "Cancel", and "Print". Below the table, there are two labels: "Total Triggered" and "Total Evaluation", each followed by a small box containing a number.

Event Description	Objective	Triggered	Evaluation Points
Verbally completes pre-taxi checklist.	1.2.3	Yes	1 of 1
Switches to Channel 20, (ATIS).	1.3.3	Yes	1 of 1
This is North Whiting Field information	2.3.4	Yes	1 of 1
Copies current ATIS information.	2.3.4	No	0 of 2
Switches to channel 1, North Whiting Ground	2.3.4	Yes	1 of 1
North Whiting Ground, Navy 6E312, taxi IFR	2.3.4	Yes	1 of 1

Total Triggered: 5 Total Evaluation: 7

Figure 19: Evaluation Results

The instructor may either 'ride along' with the student at the time of running the scenario, or analyze the student's performance later. The instructor rides along by sitting at another station that is connected via a network to the student's station. This networked machine may be anywhere that it is possible to connect via the network. So an instructor may actually be across the country while monitoring a student. The instructor may then critique the student's performance, or even take the controls from the student in order to demonstrate the maneuver that he/she wishes the student to perform.

Once the student has performed the exercise, the instructor may analyze the student's performance. The instructor can look at the summary analysis page and decide if there are any particular pieces to investigate. The instructor can use the Mission Evaluation screen to playback the entire scenario, or move to any particular piece and playback that part of the scenario specifically. This playback includes all controls by the student, including any popups and radio calls, as well as all of the student's responses.

In addition, the instructor may have entered events for the student to respond to in the

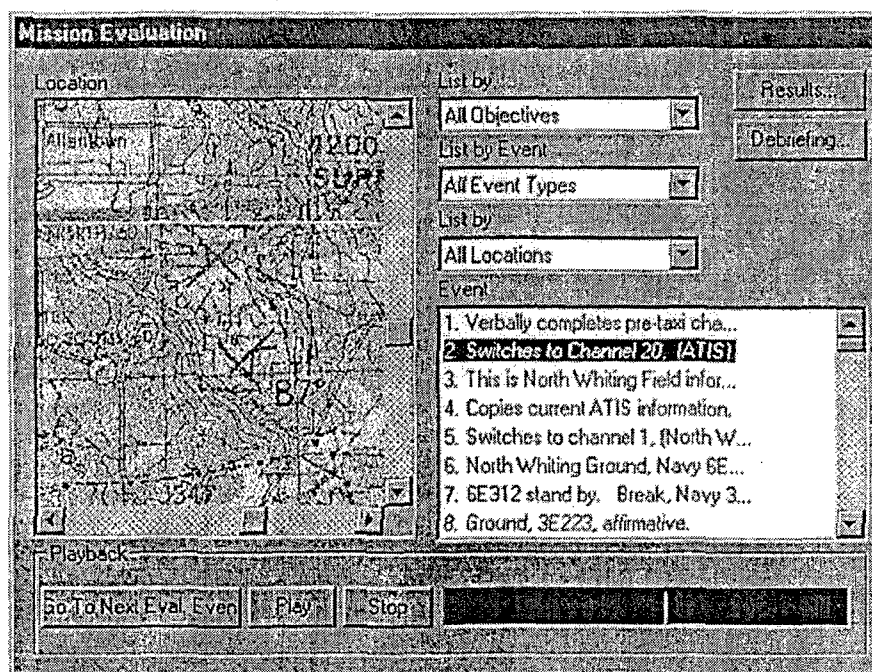


Figure 20: Mission Evaluation

scenario. These would be accessed on playback and may allow for such things as the instructor to rate the performance of a particular maneuver. Once the instructor has finished evaluating the student's performance, the summary analysis can be printed out and the students' performances can be rated.

In this manner, the instructor's grading habits may also be monitored, since any student's performance is then comparable to any other student performing the same scenario. Also, instructors may be trained by having actual instructors run the scenarios and purposely make or not make mistakes for the student instructors to grade.

5.10 Addressing Student Errors

Once OpenSkies has been informed of an error, the system is able to present the student with an appropriate branch in the training scenario. This objective defines the adaptive capability of the training system. We have developed an adaptive branching capability at this phase to present the student with different options depending on the level of capability of the student. These include such options as:

- A specifically defined branch created by the instructor.

- The system may return the student to the section of the scenario before the error was made and have the student try the process again.

- The system may ask if the student requires instruction at this point in the scenario.

The student may ask particular questions about the scenario at this point. This capability is a larger task than expected and will be developed in the Phase II effort.

The system may point out to the student that he/she missed an earlier requirement for the procedure.

Specifically, we have created the following basic events that an instructor may set up to occur in any particular scenario. These events can track interactions with the vehicle panel, the control of the vehicle, the vehicle location and particular times in the scenario. This interface works as a simple scripting interface where events are triggered or tied to one another in sequence.

For events that allow the instructor to create branches in the scenario the following events are used:

- | | | |
|---------------------------|---|----------------------------------------------------------------|
| Event Activation | - | Activate a particular event or set of events. |
| Event Deactivation | - | Deactivate a particular event or set of events. |
| Key Input | - | Activate a particular event or set of events via the keyboard. |

For marking events that are not directly tracking, such as the student examining weather maps and briefing materials, the following events are commonly used:

- | | | |
|--------------------------------------|---|------------------------------------------------------|
| Student Action by Time | - | Track a student's action at a particular time. |
| Student Action by Location | - | Track a student's action at a particular location. |
| Student Action - Independent- | - | Track the student's action after a particular event. |

For events that create communication for the student to hear and reply:

- | | | |
|----------------------------------|---|------------------------------------------------------------------------------------------------|
| Communication by Location | - | Provides a radio communication to the student once the student has entered the specified area. |
| Communication by Time | - | Provides a radio communication to the student at a particular time . |
| Communication to Student | - | Radio message dependent upon another event. |
| Play ATIS Message | - | Play weather and ATIS info. |

Radio Call From Another Aircraft - Communication from another AI aircraft in the scenario.

For events which look for student communications:

Student Message by Location - Student is expected to reply at a particular location.

Student Message by Time - Student is expected to reply at a particular time.

For events dealing with the aircraft interface:

Query Vehicle - Query parameters of the vehicle

Command Vehicle - Set parameters of the vehicle.

For events providing info to the student/instructor or asking questions of the student/instructor:

Popup FYI by Location - Popup a dialog with info upon reaching a specified location.

Popup FYI by Time - Popup a dialog with info at a specified time.

Popup Multiple Choice by Location - Popup a multiple choice question dialog upon reaching a specified location.

Popup Multiple Choice by Time - Popup a multiple choice question dialog at a specified Time.

Popup Q&A by Location - Popup a question and answer dialog upon reaching a specified location.

Popup Q&A by Time - Popup a question and answer dialog at a specified time.

For Misc. events:

System Command by Time - Toggle between cockpit and outside views, as well as end the scenario.

Building Macros

Sequences of events that get used over and over are called macros. These macros are created by building a script from the basic events and then saving this script

out to a macro file via the **Export New Event** button. This allows the instructor to cut and paste and quickly build scenarios from previous work done.

All of the predefined Event Templates are macros that are just stored in a particular hierarchy. Once an instructor has created a new macro that they wish to add, they may simply select any existing Event Template and browse to the directory where the macro was stored. The system will then prompt the scenario designer to fill in the appropriate information as shown in Figure 2.

Event Activation

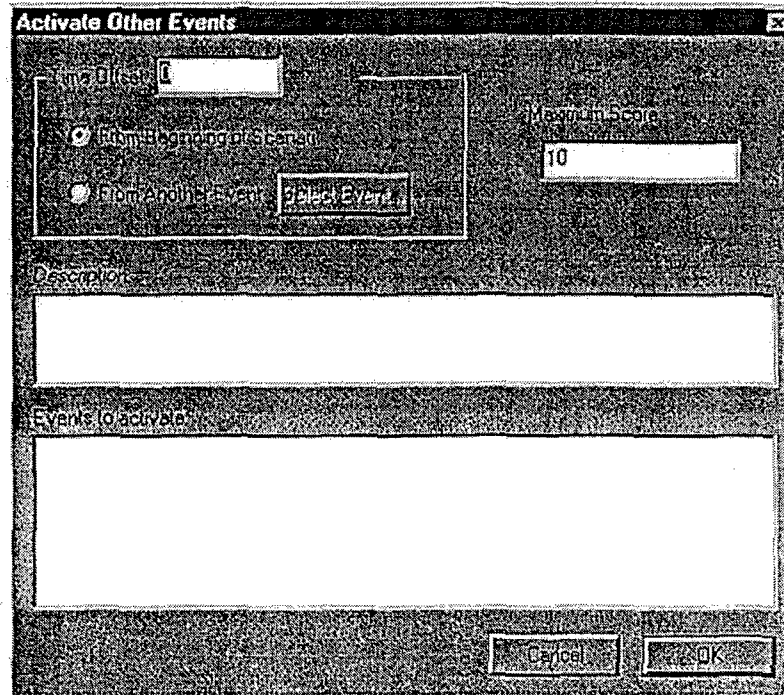


Figure 21: Event Activation Window

Time Offset:

This defines the amount of time from the beginning of the scenario or another event that this event will be triggered.

Select Event...: The instructor may choose a particular event to which to attach this event.

Max. Score:

This sets the score that the student will be given if he/she successfully handles this event.

Description:

This provides the description that is shown in the Scenario script area and in the Mission Evaluation.

Events to activate:

This specifies which events will be activated when this event is activated. The user selects from the list of events via the normal windows selection, i.e. single click, select and drag or select and ctrl select.

Event Deactivation

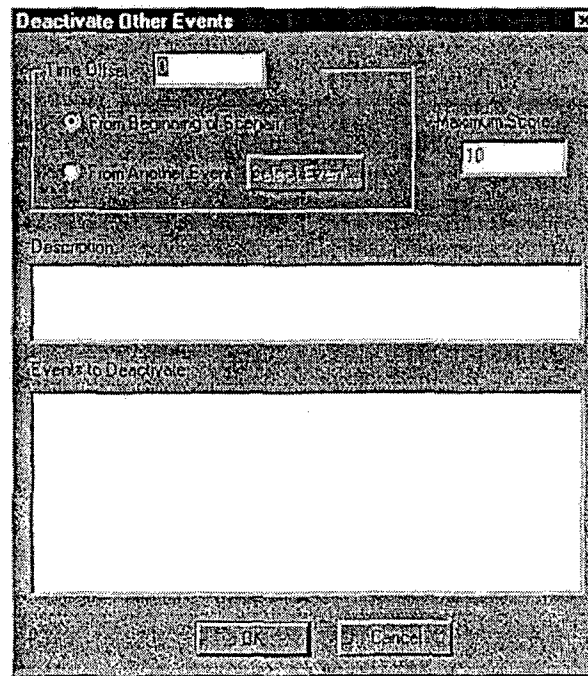


Figure 22: Event Deactivation Window

Time Offset:

This defines the amount of time from the beginning of the scenario or another event that this event will be triggered.

Select Event...: The instructor may choose a particular event to which to attach this event.

Max. Score:

This sets the score that the student will be given if he/she successfully handles this event.

Description:

This provides the description that is shown in the Scenario script area and in the Mission Evaluation.

Events to deactivate:

This specifies which events will be activated when this event is activated. The user selects from the list of events via the normal windows selection, i.e. single click, select and drag or select and ctrl select.

5.11 Example of Branching using Activation and Deactivation

A basic capability of the scenario system is that it allows scenarios to change behavior depending upon what the student does. One tool that is required for this is the ability to choose between two or more event chains depending on some condition. We call this capability branching.

Branching is shown in the Scenario script area in the Edit Scenario box via indentations and the b#> labels where # is the number of the branch.

Event activation and event deactivation provide branching capability by giving the designer the ability to "turn off" pending event elements or to cause an event to occur immediately. The following is an example of the use of the *activate* and *deactivate* script elements.

- 1) FYI Popup by Time - This script element causes a popup to occur. It triggers effects when the student clicks the OK button on the popup.
 - A) Student Message by Time (Timeout=10) - Triggers effects when the student makes a radio call. After 10 seconds, this event will time-out.
 - 01) Communication By Time (Timeout=10) - This script element causes reply #1 radio message to happen.
 - (a) Event Deactivation(1.B) - Deactivates the "no student radio call" branch.
 - (One) FYI Popup by Time - this is the last script element to occur in this mini-scenerio.
 - B) Communication By Time (Time=11) - This script element causes reply #2 radio message to happen.
 - 01) Event Activation (1.A.01.a.One) - Activates the final script element.

This mini-scenario will first bring up a FYI (for your information) popup. It will then wait for the student to make a radio call. If the student makes the radio call within 10 seconds, reply number 1 will occur followed by the final FYI popup. In

this case Branch **1.B** will be deactivated by script line **1.A.01.a**. If the student fails to make the radio call within 10 seconds, the event will deactivate itself. In this case scrip **1.B** will not be deactivated within the 11-second time offset, so **1.B** will occur, and will cause **1.B.02** to occur immediately thereafter, triggering the final script element **1.A.01.a.One**.

The end effect of all this is that the scenario will respond to the student one way if the student remembers to make a radio call, and respond in another way if the student forgets. In either case the last event is the FYI popup.

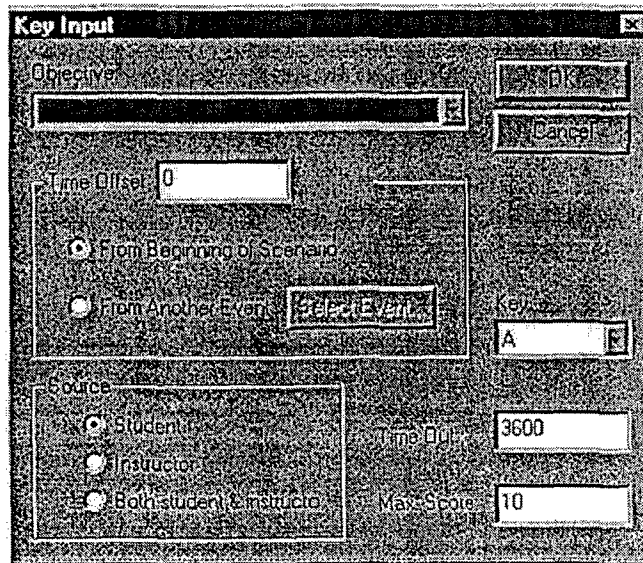


Figure 23: Key Input Window

Key Input

This event allows the instructor or student to activate a particular event or set of events in the scenario as shown in Figure 4.

Objective:

This allows the instructor to tie a particular objective to the event, such as a communications objective.

Time Offset:

This defines the amount of time from the beginning of the scenario or another event that this event will be triggered.

Select Event...: The instructor may choose a particular event to which to attach this event.

Key:

This defines the keyboard key to be used. Currently there are only 9 keys available.

Source:

This defines whether a student, the instructor, or both may activate the new events.

Time Out:

This defines the time period in which this event is active.

Max. Score:

This sets the score that the student will be given if he/she successfully handles this event.

6. Future Directions

The proposed technology will be leveraged into Cybernet's OpenSkies Massive Multiplayer training and gaming simulation business. Cybernet has developed a massively multi-player simulation technology for air, sea and land game and simulation play². While this technology was originally developed for low cost government simulation for training, the Company plans to adapt the technology to revolutionize the consumer network gaming and flight simulator industry. The Company plans to distribute its OpenSkies simulation products at retail into the market, which is currently defined by Microsoft Flight Simulator, ProPilot, and Flight Unlimited. To make a significant inroad to this market, Cybernet plans to sell the product not as an end to itself, but as the entry point to a new game playing experience.

Cybernet plans to revolutionize the gaming industry by coupling the experience of leading military commanders with the on-line flight gaming experience. The military commander will structure a training process which 'recruits' players, gives them flight training modeled after current military doctrine, and then leads graduates in a multi-player interactive war game. Because the technology is compatible with Government training needs Cybernet plans to sell this product into the Government space for integration and training purposes as well.

The current financial model for this new gaming experience indicates that it can be operated profitably in the 2nd year of operations.

The current gaming community consists of more than 7 million on-line gamers and is a \$6 billion dollar per year industry (according to Forrester Research). In addition, an on-

² The complete business plan is available upon request.

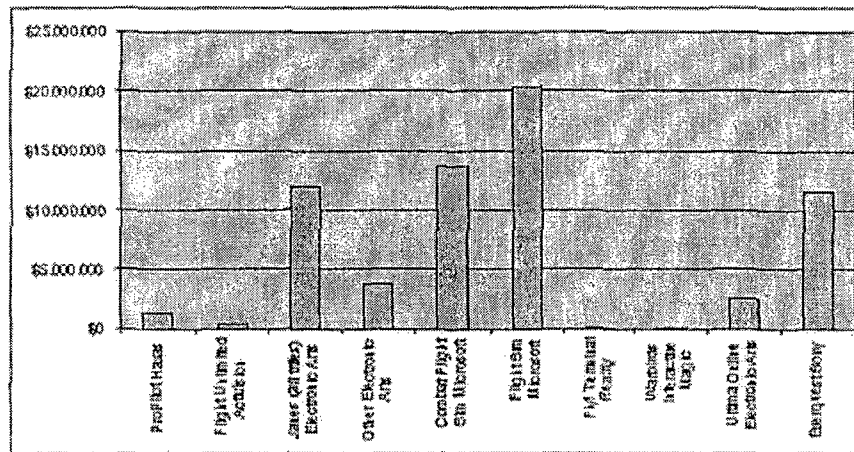


Figure 24: Summary of Sales for Leading Flight Simulators and On-line Games

7. Marketing Plan

1. Create and offer a new on-line gaming experience.
2. Focus initially on the flight simulation market.
3. Raise financing to support marketing and initial network service roll-out.
4. Start promotion for 6 months through a beta program to ensure the quality of the game experience to 250-500 players and then promote more widely to leverage good player experiences from the beta period to a wider audience.
5. Participate in game conferences and trade shows.
6. Advertise in flight simulation journals.
7. Release the first version of the game and group play experience within one year.

7.1 Target Market

Initially this will consist of the Flight simulation market. These users are the most likely to pay higher prices as most of these customers spend \$200 - \$300 a year on flight simulation gaming. We expect this to expand into the wider 3D on-line gaming market. These users typically pay a \$10.00 /month fee for access to network game services. The pricing model (nominally \$49 for the OpenSkies install CD and up to \$29 per month in network related revenue) will provide for an initial high sales market with continued revenue through the online games. This market can be expanded with more online games and more network servers to support a larger online community.

7.2 Product

This product will consist of software on a CD for Windows 95, 98, and NT attractively boxed for consumer impulse purchase. The CD will automatically install the software on the user's PC and will require that they create an on-line web account with Cybernet for the most pleasurable gaming experience.

7.3 Sales

Sales are targeted to be 100,000 units within the first 3 months of release, with sales ranging up to 2,000,000 over the lifetime of the game. Planned retention of on-line customers after the first free month of play is 30% or better. The initial beta period tested with 250-500 users will validate these expectations or force re-evaluation of the sales plan.

7.4 Budget

OpenSkies as a technology has had \$5 million invested into it as a technology platform and approximately \$3 million as a specific training/gaming system. It is our expectation that approximately \$2 million in additional development and marketing expenses will be required to move into large-scale revenue growth for this property.

8. Conclusions

This Phase I project has laid the foundation for significant advances in the use of adaptive instruction for the virtual training of helicopter pilots. The developed technology is essential to enhance the current state of training. The proof-of-concept Phase I system demonstrates the feasibility of this system. Future developments of the Phase I technology will facilitate its integration into OpenSkies to provide a full virtual training suite.